NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

WHEN IS A SIMPLE MODEL ADEQUATE FOR USE IN SCHEDULING IN MSHN?

by

Paul F. Carff

March 1999

Advisor:	Debra Hensgen
Second Reader:	Taylor Kidd

Approved for public release; Distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE March 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE WHEN IS A SIMPLE MODEL ADEQUATE FOR USE IN SCHEDULING IN MSHN?	5. FUNDING NUMBERS
6. AUTHORS Carff, Paul	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT(*maximum 200 words*)

Current resource management systems do not provide a way to use measurements taken from an application's execution on one computer to predict that application's performance on another computer. More details are needed in both their application and resource models in order to make this prediction. However, very detailed models are also not desirable. Models that are too detailed incur unnecessary overhead when values corresponding to the detail are being obtained; they are subject to higher variances; and the benefit of computing schedules using them may be outweighed by the time required to compute those schedules.

This thesis proposes a model that balances the level of detail, and therefore the quality of their predictions of resource usage, against the cost of computing schedules. To assess the quality of the proposed model, an application emulator was designed, built, and used.

The results from running the application emulator demonstrated that the proposed model is able to predict the relative resource usage of an asynchronous application that has substantially more computation requirements than communication requirements. However, an even more detailed model is needed to successfully predict resource requirements of both synchronous and communication-intensive applications.

14. SUBJECT TERMS Resource Management Systems, Operating Systems, Distributed Systems, Scheduling	15. NUMBER OF PAGES 94
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

WHEN IS A SIMPLE MODEL ADEQUATE FOR USE IN SCHEDULING IN MSHN?

Paul F. Carff
Lieutenant, United States Navy
B.S., Santa Clara University, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

Author: Paul F Carff

Paul F. Carff

Approved by: Deb A. H

Debra Hensgen, Thesis Advisor

Taylor Kidd

Taylor Kidd, Second Reader

Dan C Boger

Dan Boger, Chairman
Department of Computer Science

ABSTRACT

Current resource management systems do not provide a way to use measurements taken from an application's execution on one computer to predict that application's performance on another computer. More details are needed in both their application and resource models in order to make this prediction. However, very detailed models are also not desirable. Models that are too detailed incur unnecessary overhead when values corresponding to the detail are being obtained; they are subject to higher variances; and the benefit of computing schedules using them may be outweighed by the time required to compute those schedules.

This thesis proposes a model that balances the level of detail, and therefore the quality of their predictions of resource usage, against the cost of computing schedules. To assess the quality of the proposed model, an application emulator was designed, built, and used.

The results from running the application emulator demonstrated that the proposed model is able to predict the relative resource usage of an asynchronous application that has substantially more computation requirements than communication requirements. However, an even more detailed model is needed to successfully predict resource requirements of both synchronous and communication-intensive applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I thank most of all my loving, supporting and patient wife Laura and kids Maddie and Robby. Thank you to Debra Hensgen for making learning Distributed Systems fun.

I. INTRODUCTION

This thesis investigates the circumstances under which a particular model (described below), with a simple analytical solution, can be used to allocate resources in a restricted distributed environment. Most schedulers in today's resource management systems either (*i*) require every program that is to be scheduled to have previously been executed on every machine on which it could be scheduled, or (*ii*) completely ignore architectural and operating system differences between the various platforms. That is, these resource managers do not infer expected execution requirements on one machine from known execution requirements on another, no matter how similar or different the machines are from one another. The schedulers of the first type will refuse to assign an application to a machine on which it has not previously executed, even if the machine is identical to one on which the application has previously executed. The schedulers of the second type assign applications to machines based solely on the number of applications that have already been assigned to those machines, completely ignoring the widely varying resource requirements of different applications. This thesis examines a model which will permit a resource manager's scheduler to use information about architectural and operating system differences between machines as well as requirement differences between applications. However, this problem is large, hence, in this thesis we have chosen to restrict the environment that we consider to one that has the following attributes:

- All machines contain identical (according to manufacturer specifications) motherboards that house pentium processors, and which will, at any one time all run the same operating system (OS), either Linux Kernel 2.0.32 (Linux) or Microsoft Windows NT Workstation 4.0 (NT 4.0);
- All machines are connected by a local area network;
- Each application consists of n inter-communicating processes, each of which executes within a Java Virtual Machine (JVM);
- Each of the n processes consists of $2n - 1$ threads:

1. One **computational thread** that does not communicate with any thread in any other process and does no `reads` from, or `writes` to, disk;
2. One **output thread**, for each of the other $n-1$ processes, whose sole job is to relay information to that other process, and;
3. One **input thread** associated with each of the other $n-1$ processes, whose sole job is to obtain input from its corresponding output thread in the other process.

- The following information is known prior to execution of each process:
  1. The expected (mean) total computational time of each computational thread for each OS.
  2. The mean number of messages passed between each pair of processes.
  3. The mean size of the messages transferred between each pair of processes.
  4. The average throughput between processes on the same machine and between processes on any pair of machines for each OS.
  5. Which OS is currently executing on the machines.
- There are no applications, other than the multi-process Java applications, executing on the suite of machines.
- All processes run either in a **completely synchronous** or a **completely asynchronous** mode:

  Completely synchronous is where a group of processes are operating in lockstep fashion, each process waiting for all others to reach a certain point, at which time all of the processes of the group continue.

  Completely asynchronous is where each process operates independently from every other so that changes of state in one process are not necessarily time-related to changes of state in other processes.

The first section of this chapter describes the resource allocation, or **scheduling** problem in general. In it we attempt to motivate why a simple model, such as the one examined in this thesis, may be more preferable to a more detailed one. Our second section outlines, in general, the trade-offs that must be considered when determining the model to be used for computing a schedule. The third section describes examples of the sets of applications that correspond to the investigated execution environment. The final section outlines the organization of the rest of the thesis.

A. SCHEDULING

In order to work effectively, scheduling, which is a very important part of most every aspect of life, requires that the goals be known to a scheduler. In the Navy, officers must manage people in a way that ensures that they will complete all of their various tasks. These tasks may include training of a division, repair of equipment, or departmental work. To ensure completion of all of these tasks, the officer must assign these tasks to personnel. In order to do this, the officer must consider the goal that is to be accomplished. Does the officer want the highest quality job, regardless of how long it takes or simply that the task is completed in the shortest amount of time, regardless of the quality of work? Usually the two goals must be balanced.

To illustrate the need for sufficient information, consider the following scenario. An officer has three jobs that need to be completed. Additionally the officer is in charge of three people and schedules their work. The officer could give each person a job without regard for priority, skills, or order. Alternatively, if the officer knows that fireman Jones, the most junior person, has never done one of the jobs that is a high profile task, which must be done both quickly and with a significant degree of quality, the officer may decide to assign the job to a more appropriate person. However, even if Petty Officer Smith is the best at performing all tasks, the officer may not assign them all to her because she would not have time to complete all of them.

The scenario presented is a very simple one, but shows that, in scheduling, we must consider both affinity of tasks for resources and current loads on resources. There are many things that contribute to the decisions made during scheduling. When scheduling people, good assignment algorithms may consider many qualifications of the worker such as experience, training, and the workload already assigned to each person. Additionally, different jobs may require different equipment. For example, one job may require a special meter. The assignment algorithm may then need to know when such a tool is available. If the best tool is not available, perhaps the next best tool could be used. With the next best tool, it will take longer to do the job and this must also be considered. An algorithm must decide whether to wait for the best tool or to start the job immediately using the next best tool.

Just as in scheduling people to perform jobs, scheduling computer resources to perform tasks is a difficult problem. In fact, scheduling resources is one of the most difficult tasks that today's operating systems (OSes) perform. Programs require resources located within the machine they are executing on. The operating system manages these resources and removes the burden of scheduling from the user. For example, it transparently determines which program will use the central processing unit (CPU) every few milliseconds. In addition to performing the CPU scheduling, it also determines which program will use the network at a given instance. These decisions made by the OS happen so quickly, and are so numerous, that it would be impossible for humans to make them. We therefore rely on algorithms within the OS to handle the task of scheduling.

In the realm of interconnected or networked machines (i.e. distributed systems) we see an even larger scheduling problem. Companies now own many computers, not just one mainframe, and these computers are distributed among many buildings. At any given time, any computer may be in use or idle. Scheduling the use of these machines is important when there is contention for resources. With the decreased cost and increased capabilities of microprocessors, it is cost effective to purchase a separate computer for every person, however, with the proliferation of networks and information, resource sharing and machine or server sharing of many different types has become important.

Applications are also being distributed. For example, an application may be written to perform the same calculations on many separate pieces of data. Each piece of data can be sent to a different machine and processed. If multiple machines are available, each can run the same program on different pieces of data simultaneously. The time to complete all of the work, in that way, may be less than if each piece of data was sequentially processed on a single machine. Being able to distribute such applications over many machines, depending upon availability, enhances both fault tolerance and performance.

Many businesses are beginning to realize how wasteful the manual scheduling is that they are still using. For example, NASA's Earth Observing System (EOS) downloads terabytes of data daily. This data must be processed into four different layers of information. Each layer is processed for a specific set of user types and each layer has different requirements. Scheduling the applications and resources to process this amount of data, daily, cannot be done manually. As another example, in the military there are still many time-critical programs, such as radar processing and weapons control, which, as they are transitioning to commercial-off-the-shelf (COTS) hardware and software, will require wise allocation of computational resources. It would be much more efficient to use a resource management system (RMS) to manage these distributed resources and applications much the same as an OS manages the resources on a single machine. These RMSes will schedule the execution of programs on various networked machines. Ongoing research addresses the problem of designing ways to manage these distributed resources, just as OSes now do a good job of managing local resources.

As RMSes are integrated into daily life, information will be required to feed the RMS schedulers so that the schedulers can perform their tasks. This data must be partitioned into the right "size" pieces that the schedulers can "digest." **Granularity** is the relative size, scale, level of detail, or depth of penetration that characterizes an object or activity. It may help to think of it as: which type of "granule" are we looking at? It can refer to a level of a hierarchy, to a fineness of detail, or to an amount of information that is supplied in a description. Its meaning is not always immediately clear to those unfamiliar with the context in which it is being used. For example, in describing a car's tires, a coarse description may be to say that the car has four tires. The description of the size of the car's tires adds more detailed information and is fined grained compared to just saying that the car has four tires. We could alternatively say that the car has four tires of a specific size and made by a specific manufacturer. The information is now even more fine grained. These descriptions are relative. The more fine grained the information, the more "accurate" the description is, however, there is also "more" to the description, thus, there are more pieces that must be considered.

Accurate, fine grained information about resource capabilities and task requirements can improve scheduling. For example, consider scheduling many people to complete a large amount of work. Using fine grained information, we could generate a **perfect schedule** that matches just the right person to just the right job, ensuring that the perfect tool for each job is used. However, it may take 5 days to plan this all out. If this schedule is for only a single day's work, this will not be efficient enough. Even such a perfect schedule might not account for uncertainties of the fine grained information, such as if a piece of equipment fails or if someone takes slightly longer than expected to complete a task.

A perfect schedule is not always what is desired, as is illustrated in Figure 1. We may only want to know just enough information so that the work for the next day can be scheduled in a timely manner. In Figure 1, schedule **A** requires **SA** amount of time to calculate the schedule and **WA** amount of time to perform the work. However, schedule **B** requires **SB** amount of time to calculate the schedule and **WB** amount of time to perform the work. Comparing the two, schedule **A** takes longer to compute, but the work, **WA**, takes less time than **WB**. However, we see that schedule **B** actually completes before schedule **A** would. Thus we see that the overhead of calculating a better schedule may not always outweigh the benefits of a shorter "work-time." The same philosophy must be applied to scheduling distributed resources. Sometimes we would like to schedule the execution of applications in a way that minimizes the total time, including the time to calculate the schedule, the time to execute the schedule, and the overhead required to gather the information about the resources and tasks to be completed.

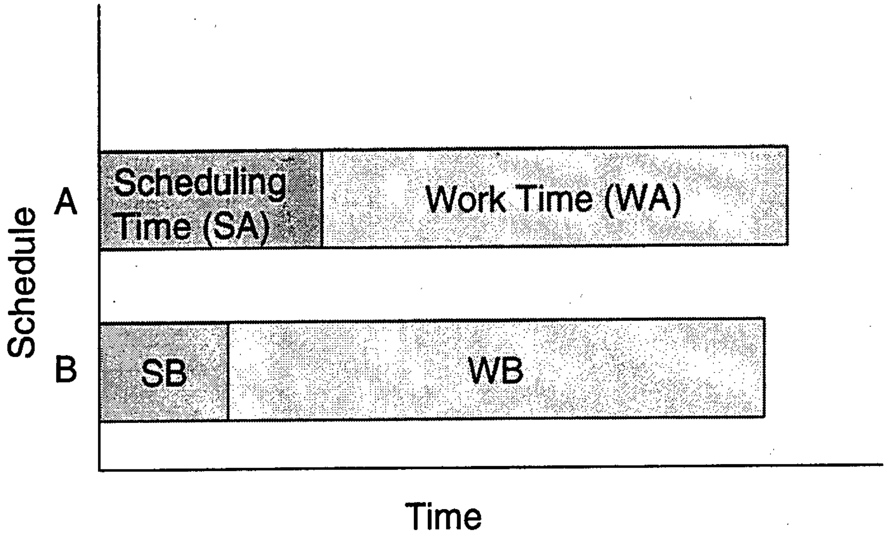

Figure 1. Comparing Schedule Time to Work Completion Time

Even though we have said that the perfect schedule is not always desired, there are situations where more effort should be imparted to determine a schedule. At times we may use a lot of information or use a more complicated method for calculating and take a little more time to determine the schedule. For example, in planning, if we know that in the future the load on our resources is going to be very heavy, we may be able to take extra time and create a better schedule for these applications in order to minimize the time it takes to complete all of the tasks, thus, planning more. But on a daily (hourly or minute) basis this may not be practical and so we would use less information and calculate the schedule more quickly, thus, planning less.

The ability of a system to recalculate a schedule is another important factor. In many environments, there may be situations where the availability of personnel and equipment change dynamically. Needs change as do priorities. These variants determine how frequently we would need to compute or recompute a schedule. If a high priority job is using a piece of equipment that fails, a new schedule would need to be calculated immediately, especially, if the replacement piece of equipment is affecting other jobs. However, a lower priority job in the same situation may not cause immediate schedule changes. Such jobs may instead be re-queued and rescheduled during the next regular schedule time. The priority of jobs will influence the frequency of scheduling and/or rescheduling.

Scheduling distributed computer resources and applications, just like scheduling personnel, is an important and difficult problem. Ongoing research is addressing the problem of designing ways to manage these distributed resources. Using fine grained information, we can generate perfect schedules. However, a perfect schedule is not always desired. An imperfect schedule may use less fined grained information and take less time to obtain, but it may be sufficient to meet all of the required and desired Quality of Service (QoS) constraints and preferences. However, there may be instances, such as in planning, where more effort should be focused on determining schedules.

B. DETERMINING AN APPROPRIATE MODEL

A schedule is determined by using an algorithm that embodies a **policy**. A policy is a definite course of action selected from among alternatives to guide and determine present and future decisions. In order to determine which particular policy to instantiate from a set of possible policies, we need to have a mechanism with which to compare policies. For any given instance, we could try all possible policies, determine the "best" policy based on a measure or metric to be optimized, and compare the results, thereby always using the "best" policy. We will show that scheduling this way is not adequate for dynamic changing environments. A much more efficient approach would be to use models of the system to compare policies in general. Although we would not be assured of always using the best policy this way, doing so provides a feasible approach that is quite commonly used. In this section we discuss how a modeler might go about obtaining the appropriate model to use for comparing policies.

We start this section with a definition of what a model is and how models are used to compare policies. The definition is then exemplified using a bank analogy. The next section discusses a method for comparing the output of a model, thus, comparing policies. In order to ensure that the policies correlate with reality, we next discuss validation methods. How to choose the appropriate model to fit the scheduling method is then outlined. Considerations must be made for the time the scheduling algorithm takes for calculating a particular schedule. The model heavily influences both the amount of time required to compute a schedule as well as the quality of the schedule produced. The next section presents an approach that is commonly used for selecting a model to match the desired scheduling performance. Finally, the set of input parameters that should be used in a model, along with their sufficiency, are discussed.

1. Models

So far we have discussed granularity and how it can affect scheduling. A **model** is an abstraction of reality that can be described by a physical representation or in the form of mathematical or logical relationships. A model is defined by its choice of input distributions and simplifying assumptions. For example, a model of a network may assume that the failure rate is 0, that latency is always 2 sec, and throughput is normally distributed around 10 Mbits/sec with a variance of 2 Mbits/sec. Models are used to compare different policies. Output values are analyzed, according to some metric, to determine which policies are better than other policies. The model must be validated against reality in some way.

As was stated previously, we want to use a model to compare two or more policies according to some metric. Figure 2 shows an example of a model, a set of policies, and performance attributes used to compare policies. The model is of a bank that has 5 branches throughout the city. The bank uses automated teller machines (ATMs) to support its customers. In order to meet the demands of its customers, the bank is trying to decide where to install 10 ATMs at its 5 branches. The number of ATMs that are to be installed at each branch defines a policy. By varying the number

Model:

A very large bank with 5 branches throughout the city. Arrival rate of Customers is a Poisson distribution with non-zero moment. Service time (time customers use the ATM) is a Poisson distribution with non-zero moment.

Policies:

How to distribute 10 automated teller machines (ATMs) among the 5 branches.

Measure to be optimized:

Minimize the time that customers wait in line to use an ATM.

Figure 2. Example of Model, Policies and Measure.

of ATMs at the branches, we change the policy. The right policy, i.e., the number of ATMs that will be installed at each branch, will be based upon a metric or measure that is used to compare the policies. The example metric that we are using is that of minimizing the time a customer waits in line for an ATM. Specifying the input sets of a model, within the limits or boundaries of a policy, will provide distributions on the output. By analyzing these results, using the metric of interest, policies can be compared, and the modeler can then determine whether one policy is better than another policy.

After the policy (or set of policies) has been defined, we can run simulations, or obtain analytical closed-form or iterative solutions (see next section) and measure one of several performance measures. Distributions of performance attributes for different policies are compared. This comparison will identify those policies for which the distributions differ significantly. Many methods exist for comparing distributions.

One example is the χ^2 **Goodness of fit test**. This test compares a distribution of data to a known distribution, or another set of data, and determines whether the two are the "same" up to a **confidence percentage**. This confidence percentage is a measure indicating how closely the two distributions are correlated. It is a threshold of how sure we want to be if and when we say that two distributions are the "same." The details of this test can be found in many references [Ref. 1].

Once the data have been collected, and the policies evaluated, based on a comparison of these data, the modeler must determine whether these results are correlated with reality. All of the simulation and collection of data will mean nothing if they cannot be used to accurately predict real performance. **Validation** is concerned with determining whether the model is an accurate representation of the system under study, i.e., reality, and is something that should be done throughout the entire modeling process. A model can only be an approximation to the actual system, thus there is no such thing as an absolutely valid model. The model should always be developed for a particular set of purposes.

The most definitive test of a model's validity is establishing that the output data closely resembles the output data that would be expected from the actual system. If a scaled down version of the actual system exists, then a scaled down model of the system should provide highly correlated output data compared to those of the existing system itself. That is, if the set of data from the model and the scaled down version correlate within the required accuracy, then the model of the existing system is considered "valid."

2. Choosing a Good Model

Figure 3 demonstrates that there are many approaches that can be used to determine schedules. The goal of scheduling in an RMS is to obtain a mapping of a set of applications to a set of machines and an ordering of these applications that optimizes some metric. An example metric that we used in this thesis is minimizing the time that the last task, of a set of tasks, takes to complete. Determining the

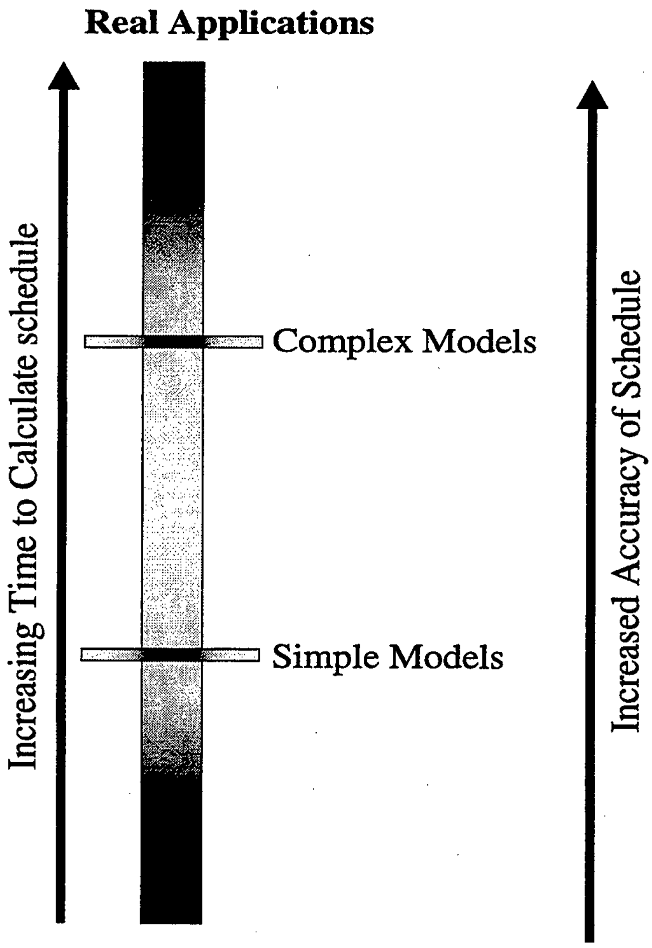
Real Applications
Increasing Time to Calculate schedule
Complex Models
Simple Models
Increased Accuracy of Schedule

Figure 3. Scale of possible ways to schedule

perfect schedule for a set of tasks may require actually executing the tasks on all of the possible combinations of machines several times, computing the average of the completion times, and selecting the schedule that performed the best on average. This approach, which corresponds to the top of Figure 3, would only be viable in a limited set of circumstances and, even then would only be useful for planning. This approach requires complete knowledge of which applications will be requested at what time. Also, we must know exactly which resource set will be available. This approach is clearly impractical in most circumstances where the set of tasks to be executed or resources available, or both, changes dynamically.

A better method for calculating a schedule most of the time would be to perform a simulation of the various schedules. Simulation could use very complex and accurate models for both the equipment and applications or very simple ones. The level of model detail affects the length of time to compute a schedule. The modeler must determine which aspects of reality actually need to be incorporated into the model and which can safely be ignored.

Figure 4 demonstrates a process that the RMS designer can use to determine how detailed their model should be as well as whether they need to use simulation. Often, a set of equalities and inequalities can be derived from a model. Many times there is a closed-form solution to these equations (and inequalities) and thus, as the first decision diamond illustrates, analytical solutions may be used to determine a good schedule. This method is very fast, however, if there is no closed-form solution there may still be an iterative solution that converges. Iterative methods are not as fast as closed-form solutions. However, an iterative approach is still much more practical than actually running all possible combinations of the real programs.

If a model has neither a closed-form solution nor one that converges, the modeler must then refine the model. The modeler must determine whether the model should be simplified to one that has either a closed-form solution or one that converges. If so, then the modeler will go back to the beginning and try again. If such simple models do not correlate to reality, however, the modeler will need to use discrete event simulation (DES).

If DES must be used, then the modeler must decide whether to use a deterministic or Monte Carlo simulation. When a deterministic DES is used, the simulation is executed for a finite number of input sets and the modeler makes a decision based upon the output. If there are an infinite number or even a very large finite number of input values that would require too much simulation time, then a Monte Carlo simulation is sometimes required, though other alternatives are sometimes available. For example, when Texas Instruments tests its chip designs for stuck-at faults, they

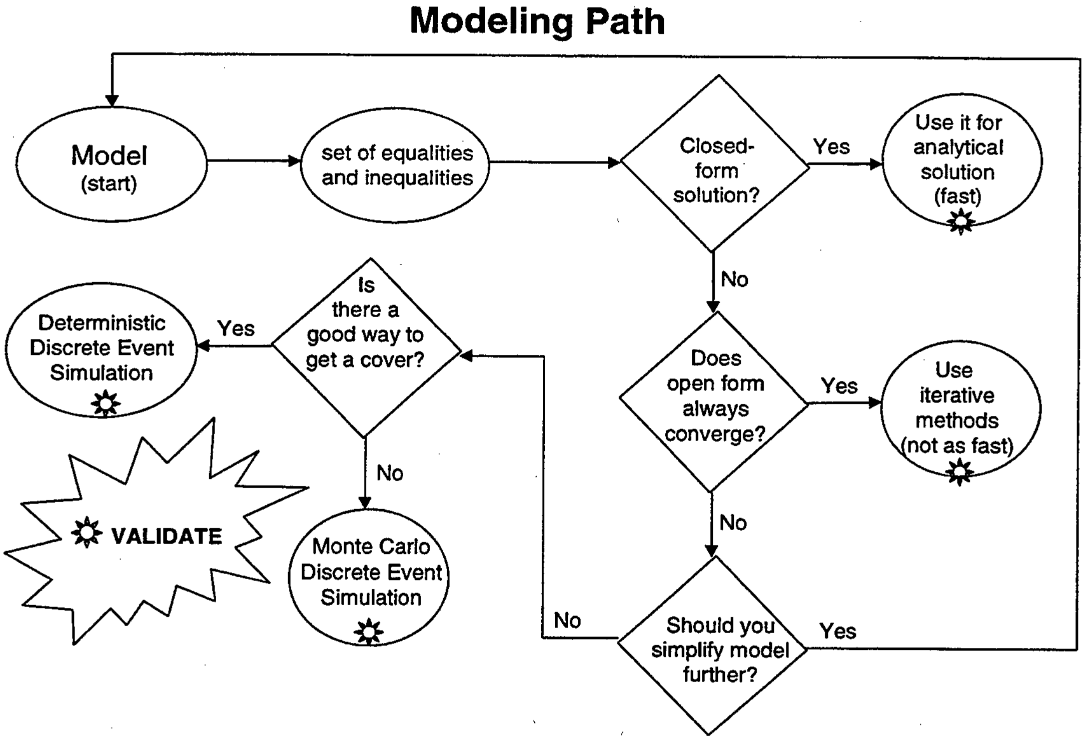
Modeling Path
Model
(start)
set of equalities and inequalities
Closed-form solution?
Yes
Use it for analytical solution (fast)
No
Does open form always converge?
Yes
Use iterative methods (not as fast)
No
Should you simplify model further?
Yes
No
Is there a good way to get a cover?
Yes
Deterministic Discrete Event Simulation
No
Monte Carlo Discrete Event Simulation
VALIDATE

Figure 4. Pathway to selecting a model.

cannot afford the time (years) to test for all possible inputs. Instead, they have developed theoretical methods that allow them to identify a small set of inputs which reveal a large percentage (greater than 95%) of such faults. However, when the input set is very large and such domain-specific techniques are not available, Monte Carlo simulations must be used. A Monte Carlo simulation causes input values to be selected according to the model's input parameter distributions. Because a Monte Carlo simulation uses distributions, it will need to be run multiple times in order to get a statistically valid result.

Even if a model uses either a closed-form solution or an iterative approach, it will not be sufficient unless its results are highly correlated with reality. If these methods produce results that do not match reality, then they are not good models. Comparing these methods with reality requires the definition of a measure to determine when the results are "good enough." The measure of importance to us will be covered in chapter III. Because comparison with reality may invalidate the use of a particular model, determining the proper model is itself is an iterative process.

C. APPLICATION MODEL APPLICABILITY

The class of applications that fit the model in the introduction is large, although, it does not cover all applications. In this section, examples of classes of such applications are described. Additionally, an example of a class of applications that do not fit the model is given.

When describing the example application classes, both existing programs and programs that can be expected to be written in the near future are included. The two example application classes that we describe are parallel discrete event simulation and physical and biological problems that are solved using iterative numerical methods. Both of these large application areas are currently being used in both fully synchronous and fully asynchronous modes. In the fully synchronous mode of discrete event simulation, processes synchronize with each other's **virtual time**. In the fully synchronous mode of iterative numerical methods programs, processes execute in a lock step mode, exchanging updated values after each iteration. Asynchronous discrete event simulation is known as the **time warp paradigm** [Ref. 2] [Ref. 3], where each process simulates at its own rate, rolling back in virtual time only if it receives an event in its past from another process. The asynchronous version of the numerical methods applications leverages the fact that many of these numerical algorithms will converge quickly even if many processes are using some values from previous iterations. Although not all of the programs in both of these classes have been ported to a multi-threaded implementation of Java, as both the speed of interpreters and the speed of processors continue to accelerate, it is quite clear that such applications will soon be written.

D. ORGANIZATION

Chapter II of this thesis will discuss related work in the area of resource management systems for distributed computing systems, and in particular, their contribution to the area of scheduling. It also discusses some other recent research in distributed computing which is relevant to this thesis. Chapter III discusses our analytical model and our emulator, which we use for validation. The description of our approach is followed by chapter IV, which presents our experiments and analyzes our results. A final summary is given in chapter V.

II. RELATED WORK

In this chapter we review work that is related to various aspects of the research presented in this thesis. Out motivation for the research described in this thesis is that existing resource management systems provide insufficient detail in their resource and application models to enable them to obtain good schedules in many situations. However, given that the number of different areas that must be researched to build an effective resource management system (RMS) is tremendous, it is not surprising that the most effective RMS projects have been those that concentrated on one or a few of the important areas. For example, the Condor project [Ref. 4] [Ref. 5] focused on providing mechanisms for transparent process migration in Unix. Odessey [Ref. 6] focused on devising an architecture that maximizes the agility of mobile applications. In this chapter, we only provided an expanded discussion of two resource management projects, SmartNet and MARS, that contributed significantly to the current state of scheduling for heterogeneous systems. SmartNet is the predecessor of MSHN, the RMS research project to which this thesis contributes. The first section of this chapter reviews some relevant RMSes.

RMS architectural projects are not the only projects that have a relationship to the research described in this thesis. Some research projects have focused on one particular aspect, such as monitoring, and have not been concerned with the relationship of that aspect to the entire system. For example, the Network Weather Service research project from the University of California at San Diego focuses on the problem of monitoring as does MSHN's wrapppers. The second section of this chapter discusses the related monitoring projects.

Finally, some related work has been done in application specific scheduling, particularly the AppLeS project. The last section of this chapter reviews this project.

A. RESOURCE MANAGEMENT SYSTEMS

Various methods have been employed to improve the assignment of resources to applications. Various metrics are used to measure the improvement. This section reviews a few of the methods currently in use to improve performance.

1. SmartNet

SmartNet is a scheduling framework for heterogeneous systems developed by the Heterogeneous Computing team at the US Navy's facility at the Naval Command, Control, and Ocean Surveillance Center (NCCOSC) for Research, Development, Testing and Evaluation (RDT&E) in San Diego [Ref. 7]. SmartNet's goals are to maximize computing power and increase the throughput of a set of jobs by better leveraging existing resources [Ref. 8].

SmartNet consists of four processes:

1. The SmartNet Controller process interfaces with resources. The resources that the controller process manages include physical ones such as machines and virtual ones such as RMSes (including additional instantiations of SmartNet).

2. The SmartNet Scheduler contains both optimization and scheduling algorithms. The scheduler includes Exhaustive, Greedy, Evolutionary, and Simulated Annealing algorithms. The Scheduler is designed so that new algorithms may be easily integrated as they become available or necessary.

3. The SmartNet Database is an ASCII text file containing information about sites, groups, machines, jobs, and model-machine pairs. SmartNet tracks expected time for completion (ETC) data in the model-machine listings that the Scheduler uses to create near optimal job-machine assignments.

4. The SmartNet Learning and Accounting process tracks and reports rogue processes. A rogue process is one that exceeds its ETC by more than a certain tolerance and endangers the schedule developed by the Scheduler. Additionally, the Learning and Accounting process gathers experimental data on the actual run-time of jobs. It uses this data to update the ETC field of the model-machine listing in the Database.

The scheduler used in SmartNet uses existing data already obtained about applications that are to be scheduled. It must know these characteristics *a priori* in order to produce an effective schedule. If a new resource is added to SmartNet's virtual machine, an application's performance data for that new resource must be added to the SmartNet database in order for the SmartNet scheduler to generate a good schedule. The SmartNet scheduler assumes that it is controlling all allocation of the resources.

2. MARS

The **Metacomputer Adaptive Runtime System** (MARS) is a framework for minimizing the execution time of distributed applications on a wide area network (WAN) metacomputer [Ref. 9]. A metacomputer is an abstract computer consisting of networks of computers, workstations and supercomputer modeled as a single powerful resource. However, compared to traditional computing resources, a metacomputer may be very dynamic. Its components can vary in time, in performance, and in connectivity.

Work-load balancing in based on dynamic information about the processor load and network performance like SmartNet's. MARS uses accumulated statistical data on previous execution runs of the same application to derive an improved task-to-process mapping. Unlike SmartNet's non-instrusive data collection, MARS, data is collected by a preprocessor inserting extra statements into the user's application code, so that statistical data on the program is collected at run time.

B. MONITORING

Any model used to predict the performance of an application requires some sort of data as input. The quality of the data will affect the results as much as the model itself. In this section we survey work in progress to refine the quality of data that are used as input values to models.

1. Network Weather Service

The Network Weather Service (NWS) is a tool for predicting computer and network performance for use by meta-computing applications [Ref. 10]. The NWS takes periodic measurements of deliverable resource performance from distributed networked resources, and uses numerical models to dynamically generate forecasts of future performance levels. Resource allocation and scheduling decision can then be based on these predictions of the performance.

Two of the many parameters used in predicting performance of an application are TCP/IP end-to-end throughput and latency. Using adequate values for these parameters are required to support scheduling. The NWS uses multiple methods for predicting these values [Ref. 11]. Which method used is determined by monitoring the accuracy of the each model (using prediction error as an accuracy measure) and using the one exhibiting the lowest cumulative error measure at any given moment.

These monitored values are continually updated. Thus, when a schedule needs to be calculated, these values are retrieved from the NWS and are used as input parameters for the model. Matching the time that the prediction is valid to the model run time will optimize the value of a schedule. If the predicted value is only good for 10 seconds but the model runs for 30 seconds, then the results returned from the model are not as accurate as we would like.

Choosing the correct predictor method for each resource that the NWS monitors is difficult, so all predictive methods are maintained simultaneously. However, for dynamic scheduling of applications to know this information is not usually the case and the current method employed by the NWS is probably the best choice. Which ever method is used, as long as the quality of the data obtained is improved, without incurring too much overhead, the prediction made by the model will improve.

2. MSHN Wrappers

MSHN requires the gathering of resource usage information for applications that run within the MSHN system as well as resource status information within the scope of the MSHN scheduler. The MSHN scheduler uses this information to make scheduling decisions. The methods used to implement the gathering of this information are subject to three constraints:

1. The implementation must not require any changes to the operating system.
2. Modifications to the application code must be minimized.
3. The overhead imposed by the information gathering mechanism should not be excessive.

The gathering of data is accomplished through a method that intercepts systems calls [Ref. 12]. The client library is linked with the object code of the application that is to be monitored prior to run-time. During run-time the client library gathers information on an application's resource utilization by intercepting system calls and through the use of operating system functions. Additionally, the client library determines end-to-end perceived status of the resources that the application uses. Using the client library, the network resource data is collected passively.

C. APPLICATION-SPECIFIC SCHEDULES - APPLES

Application level scheduling (AppLeS) agents are used by the University of San Diego (UCSD)

- to schedule tasks and communication on heterogeneous computation and network resources which exhibit individual performance characteristics, and
- to schedule computation and communication resources which are shared, under the control of different local schedulers, and/or located in distinct administrative domains [Ref. 13].

These agents are based on an **application scheduling paradigm** where everything about the system is evaluated in terms of its impact on the application. Each application has its own AppLeS, whose function it is to select resources, determine a performance-efficient schedule, and implement that schedule with respect to the resource management infrastructure of the metacomputing system.

An AppLeS agent is organized in terms of four subsystems and a single active agent called a **Coordinator**. The four subsystems are

- the **Resource Selector**, which chooses and filters different resource combinations for the application's execution,

- the **Planner**, which generates a resource-dependent schedule for a given resource combination,
- the **Performance Estimator**, which generates a performance estimate for candidate schedules according to the user's performance metric, and
- the **Actuator**, which implements the "best" schedule on the target resource management system(s).

AppLeS requires either previous knowledge about the application's performance on the particular resources on which it is to be scheduled, or for the user to supply this information via the User Interface (UI).

D. SUMMARY

If a new application, that has never been run on a subset of the available resources, is to be scheduled by current systems, then the user must supply the data, or use default values. As more applications and resources are added to a distributed system, the longer it will take to determine the eventual best schedule since more combinations of actual program executions will be required, i.e., every program (old and new) will have to be run on the new resources and new programs will have to be run on the old resources. Then, and only then, will there be complete information for determining the best schedule. The next chapter describes our approach to develop a model with the proper granularity for use in scheduling. Our model is a step in the direction of solving this problem in current systems.

III. APPROACH

As stated in the introduction to this thesis, current RMSes and Common Object Request Broker Architecture (CORBA) Object Request Brokers (ORBs) take one of two approaches. The first approach, used by the most intelligent ORBs, assigns tasks to machines based solely on the number of tasks already assigned to that machine. That is, they do not account for either the differing resource requirements of different requests, nor for the differing capabilities of the various machines. The other approach, which used by SmartNet [Ref. 7], uses explicit knowledge of the capabilities of a particular machine to execute a particular application. Although this approach is much better than the first, if the capabilities of each machine to support every request is known *a priori*, it cannot assign applications to machines based upon known similarities or differences. In the MSHN project, we attempt to predict the quality of service that a particular machine will be able to provide for each request based upon the resources of the machine, the current expected load on these resources and the resource requirements of the requests. This thesis grew out of MSHN's goal to determine the granularity with which (*i*) resource requirements of applications, (*ii*) resource capabilities of machines, and (*iii*) resource allocation policies of OSes must be modeled. However, this problem is huge. A major goal of this thesis research was to scope this problem so that this thesis, and tools developed to perform this research, would contribute in a meaningful way to answering the question. Therefore, we have scoped this problem as described in the introduction to the first chapter.

This chapter is divided into three major sections. The first section discusses, both a generally and a specifically, the issues that must be considered when determining the appropriate granularity to use in a model for resource allocation. The second section describes a model that, while fairly detailed, possesses a closed-form solution. The third section describes our approach for determining the conditions under which our model is sufficient for resource scheduling.

A. MODEL GRANULARITY

The granularity of information used in a model is the result of a trade-off. Initially one could assume that the finest granularity information would always be the best. However, we show in this section that fine grained information has characteristics that can make it undesirable for use in scheduling algorithms.

1. Overhead

Fine granularity resource usage models require a large amount of overhead to collect the volume of information required to instantiate the model. If the resource usage requirements could be determine prior to compile time, then, if the application was run many times, the extra overhead at compile time, which would be incurred only once, would be offset by the increase performance every time the program is run. However, if the data must be collected, analyzed, and a decision made every time the program is executed, the overhead can be substantial. Additionally, run-time collection may perturb the measured value. For example, measuring the throughput of a network using a method that loads the network will measure an available throughput that is less that what is actually available.

2. Variance

Finer granularity information has higher variance than coarser granularity information. Using the Central limit theorem it can be shown that any distribution (e.g., exponential), if summed, would approximate a normal distribution [Ref. 1]. A side-effect of this theorem is that the resultant standard normal variance is less than the sum of the variances of the distribution. In other words, the coarser grained information has less "error" and can result in a better schedule.

3. Complexity

As discussed in Chapter I, finer granularity models require more complex scheduling algorithms. A simple closed-form solution uses coarse granularity information. However, as the model integrates more and more data, in detail or in volume, then, in order to process the added amount of information, the resource allocation model needs to become more complex. These solutions range in complexity from iterative solutions to Monte Carlo simulations.

4. Criteria for Determining Required Granularity

If there were not issues, such as the three described above, finer granularity models would be most preferable. However, such issues are often overshadowing issues, so we must address the question of how coarse the granularity of the model can be and still provide the basis for a good resource allocation algorithm. The answer depends upon what we mean by good resource allocation and is discussed in the next section.

Different granularities may be useful for different purposes. The first goal of this research was to determine a simple model that may be good enough to use when assigning resources to processes. We considered two different interpretations of what it means to be "good enough:" relative performance and absolute performance. We now discuss the meaning of these interpretations.

a. Relative Performance

In Figure 5, we show both the predicted and the actual run times of two schedules, A and B. Comparing actual completion times, A1 and B1, with the corresponding predicted completion times, A2 and B2, we notice huge differences between the actual and predicted run times. However, we also note that if we had only the predicted completion times A2 and B2, we would expect that schedule A is better than schedule B and indeed it is. In this case, if we only need to choose between **A** and **B**, we would say that the model used for prediction is sufficient. An example of such a model would be one that assumes that the overhead incurred by each schedule is zero when it is actually almost a non-zero constant over all schedules. In this case we will say the model is sufficient to predict the relative performance.

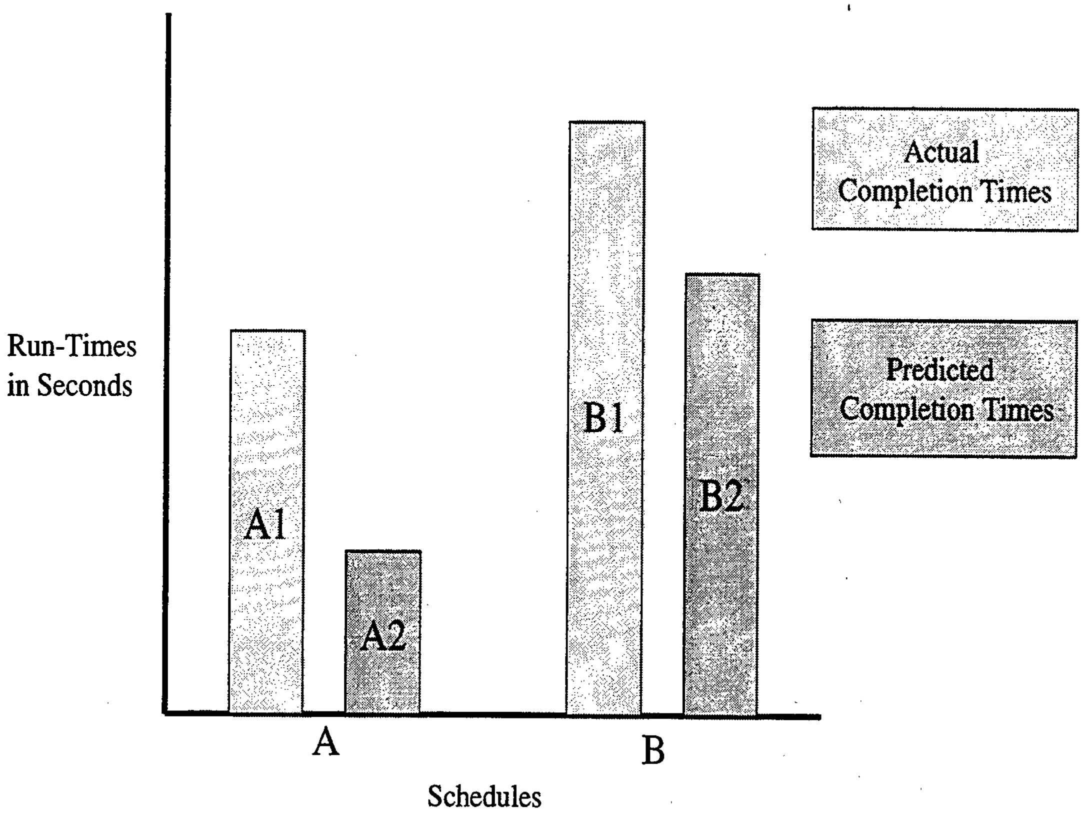

Figure 5. Example Actual and Predicted Completion Times.

b. Absolute Performance

We must be careful not to use the types of predictions discussed above in other situations. For example, suppose we assign resources according to schedule **A** for some set of requests, and assume that schedule **A** assigns all processes to one machine and schedule **B** assigns all processes to another machine. As soon as the processes begin executing, we may receive another request to execute another set of the same processes. The predictions shown in Figure 5 indicates that we should choose the same schedule, **A**, for the new set of processes (because twice the height of **A2** is less than the height of **A1**). However, we see that the better choice would be to use schedule **B** for the second set of processes (because the height of **B1** is less than twice the height of **A1**).

B. MODEL

Although we recognize that there are more important measures, we have scoped this research to only account for a simple one: time at which the last process, given a particular schedule, completes. The purpose of our simple model is to allow us to accurately estimate this measure. Using a simple model limits the possible number of applications that can be accurately modeled. However, we want to choose a model that applies to a wide variety of applications, yet is detailed enough to give adequate results for use in scheduling. Therefore, we define a set of parameters that will enable us to specify the resource usage of an application, while simultaneously allowing the usage patterns to remain flexible.

A class of applications can be defined by a set of compute characteristics. Compute characteristics are the set of parameters that influence the resource usage patterns of the application [Ref. 7]. We can divide the resource patterns of a program into two categories: its computation load and its communication load. More specifically, we can define the application's computation load as the amount of time it takes the application to perform the computationally intense part of its work. The communication load can be defined as the number and size of messages that the application must send. These parameters (computation "time", number of messages sent and received , and size of messages) can be used to characterize many types of applications. We recognize that communication time and computation time can, and do, overlap. We do not include this overlap in out model, although we address this again in Chapter V.

1. Computation Time

The computation time is the amount of time the program uses the CPU. This is not the total time that the program takes to run. The total run-time is a combination of computation time and communication time. We assume we know *a priori* the expected time that each application will use the CPU, i.e., the computation time. If we obtain the expected computation time on one machine, it is a trivial matter to

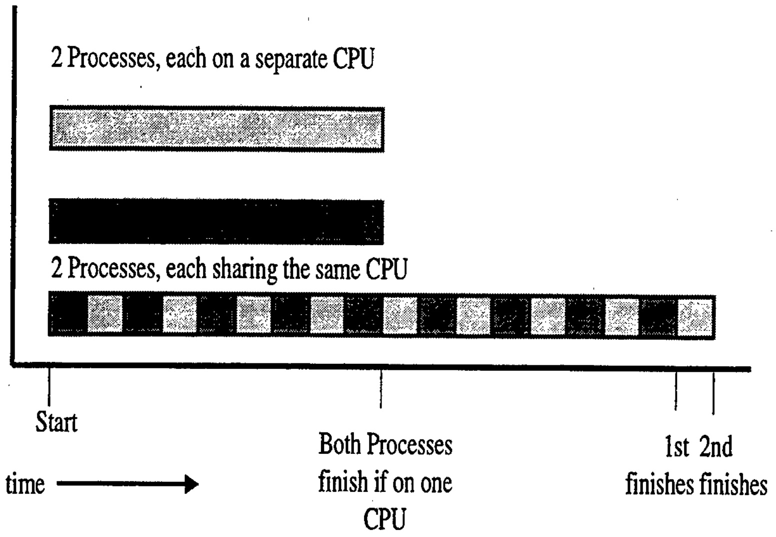
2 Processes, each on a separate CPU
2 Processes, each sharing the same CPU
Start
time
Both Processes finish if on one CPU
1st finishes
2nd finishes

Figure 6. Effect on "Wall-clock" time due to two processes sharing a single CPU

predict the expected computation time on a processor with the same instruction set but different speed CPU. If they have a different instruction set, an averaging process using some benchmark suite can be used to translate between processors [Ref. 14].

Since modern OSes alternate multiple programs on a single CPU, we must account for the sharing of this resource. A program sharing the CPU, will take more "wall-clock time" to complete the job. No more CPU time is required for an individual program because the program shares the CPU. However, one program will run for a short time and then another will run for a short time. If we consider only two programs, then after the second program has run for a short time, the first program will get the CPU back again. This time-sharing has the effect of dilating the program wall clock time to a maximum of the sum of the two individual stand-alone run-times as shown in Figure 6. Although we recognize that context switching time is non-zero, we have assumed it to be zero in this model [Ref. 15]. We further discuss the affect in Chapter V.

We now discuss, given a schedule, how to estimate the **computation wall clock time (CWCT)**. The CWCT of a process is different from its computation

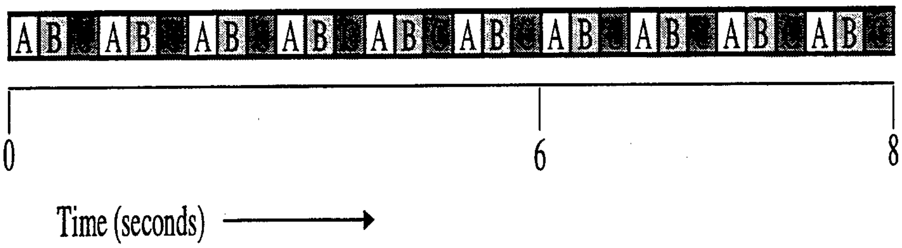

Figure 7. Example of Dilation of a Process Computation Time Due to Sharing

time if other processes are contending for the CPU. We now develop an algorithm for estimating CWCT. CWCT is the difference between the time that a process finishes using the CPU and the time it starts, neglecting communication and context switching time.

Our algorithm, which can be used to estimate the CWCT of any process, identifies the intervals during which that process is sharing the CPU with other processes. For example, suppose processes **A**, **B**, and **C** are assigned at time 0 to a machine with a single CPU. Suppose further that **A** requires 2 seconds of CPU time, **B** requires 3 seconds, and **C** requires 5 seconds. Suppose that we are estimating the CWCT for **B**. All three processes are initially sharing the CPU until the shortest one, **A**, finishes. Therefore, the first interval would last approximately $3*2 = 6$ seconds (See Figure 7). At time 6, 2 seconds of **B**'s CPU time, out of a total of 3 seconds, have already been accounted for. Therefore, **B** only needs 1 second of CPU, but it is still sharing with **C**. Similarly, **C** still needs 3 seconds of CPU time. Because **B** is the shorter the two, it will complete first after $2 * 1 = 2$ seconds, or at time 8.

We now present a formal algorithm for estimating the CWCT of a process, P, given that K is a list of all processes that have been assigned to the same machine as P and are sharing the same CPU as P[1].

[1] Note that K is a list of processes (such as A,B, and C above) and variable *shorterList* in the algorithm is a list of CPU times.

```
function CWCT(process P) {
    CWCT = 0;
    left = 0;

    shorterList = //Initialized to an ordered list containing
                    the CPU times of all processes with a CPU time
                    less than or equal to P -- P must be the last
                    element of the list//;

    numberLonger = sizeOf(K) - sizeOf(shorterList);
    while(not empty(shorterList)) {
        n = sizeOf(shorterList);
        right = deque(shorterList);
        CWCT = CWCT + ((n + numberLonger)*(right - left));
        left = right;
    }
}
```

We note that each iteration of the while loop identifies an interval, which extends from left to right. Also, $n + numberLonger$ is the number of processes sharing the CPU during this interval. We also note that as a side-effect of estimating the CWCT for the longest process, the CWCT for all processes is estimated; hence, this algorithm would only need to be executed once.

2. Communication Time

Communication time includes time to send and receive messages and time spent waiting, i.e., for buffers or for incoming messages. Our model assumes that waiting time is 0. To do otherwise would either cause our model to be of such fine granularity that simulation would most likely be required or we would need to know an average waiting time distribution per message. However, we were unable to identify a good representative distribution in the literature.

Neglecting waiting time, communication time is divided into three parts:

1. The time spent preparing the message by the sender which we call the sender's preparation time;
2. The time transmitting and propagating the message; and

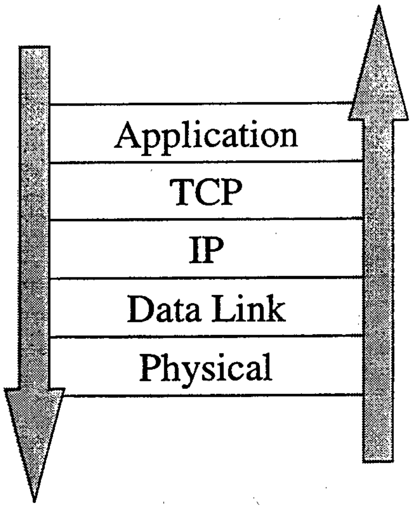

Figure 8. Example of a Network Protocol Stack.

3. The time spent processing the message by the receiver which we call the receiver's processing time.

Figure 8 demonstrates how a message must travel down and up the various layers of the protocol stack, from the application down to the network hardware on the sending end, and up from the network hardware to the application layer at the receiving end. At each level there may be some processing involved. At the sending end, for example, a particular layer may incur the overhead of adding a header of information. On the receiving end, the same layer must also incur some similar processing overhead due to unpacking and decoding the header. We call the total time required for the processing of each message the **latency time**. Since this processing of messages requires the use of the CPU, the amount of time in 1.) and 3.) would need to be dilated as discussed above for computation time.

We use the following method to estimate 1.) and 3.). On completely quiescent machines and networks we send a large number of very small messages, (e.g., 10,000 one byte messages), to a receiver and have the receiver immediately send the message back to the sender. We record the time required to complete these transmissions.

This recorded time is the time it takes for all of the messages to travel down the protocol stack on the sender's side, propagate across the medium to the receiver, travel up the protocol stack on the receiver side and to return the message back to the sender. Thus, if we assume that the latency time in each direction is the same, then for each message, we incur two latency times. Since the message size is small, and our model assumes a fast local area network, we assume the transmission time is zero. So, we can estimate the latency time by first dividing the total time to send and receive all messages by two, to obtain the total one-way latency time. Then, to produce the latency time per message, we divide the total one-way latency time by the number of messages sent. For example:

```
Number of messages  =       10000
Time to send and receive = 7400 ms
Adjust for two latency times per transaction:
                           7400/2 = 3700 ms
Latency per message =      3700/10000 = .37 ms
```

Tables I and II shows the values we obtained.

Latency Time (ms)	NT		
	Gratian	Tiberius	Pius
Gratian	0.32	0.36	0.36
Tiberius	0.36	0.32	0.36
Pius	0.36	0.36	0.32

Table I. Summary of Latency Times for Windows NT 4.0

So far we have discussed how we estimated parts 1.) and 3.) of the communication time above. We now discuss how we estimated 2.), the time required to transmit and propagate a message from a sender to a receiver. As stated in the introduction, our model assumes the average number of messages and average size of those messages, thus, the expected total amount of data, that is to be transmitted over each link is known *a priori*. A **link** is defined as a connection between two processes such

Latency Time (ms)	Linux		
	Gratian	Tiberius	Pius
Gratian	0.44	0.44	0.44
Tiberius	0.44	0.44	0.44
Pius	0.44	0.44	0.44

Table II. Summary of Latency Times for Linux

as a network or shared memory inside a machine. We also assume that the maximum throughput of the link used to transmit can be measured on a quiescent network and machines.

Figure 9 illustrates communication links between three processes, **P1, P2**, and **P3**. We define these links between specific processes, such as from **P1** to **P2**, as a **channel**. We notice that the communication between **P1** and **P3** occurs within the same machine, thus this communication uses the shared memory of the machine, whereas, the communication between **P1** and **P2**, and **P3** and **P2** occurs over a network link since **P1** and **P3** are located on a different machine than **P2**. Figure 9 also shows that the various links are shared just as the CPU is shared and thus we must dilate the communication time just as we did the computation time. The **dilated communication time (DCT)** is estimated in a manner very similar to the CWCT. Again, although contention costs are non-zero, and as in CPU contention, network contention per use increases with the number of "users," we assume it to be zero.

We now exemplify how to estimate the DCT. We assume that we have measured the throughput between machine **A** and machine **B** of Figure 9 as 1 Mbps and the internal throughput on machine **A** as 8 Mbps. Further, we assume that process **P1** has 10 MBytes of data to send to **P2** and 10 MBytes of data to send to **P3**, process **P2** has 20 MBytes of data to send to **P1** and 20 MBytes of data to send to **P3**, and finally, process **P3** has 15 MBytes of data to send to **P1** and 15 MBytes of data to send to **P2** as shown in Table III. If there was no sharing of resources then,

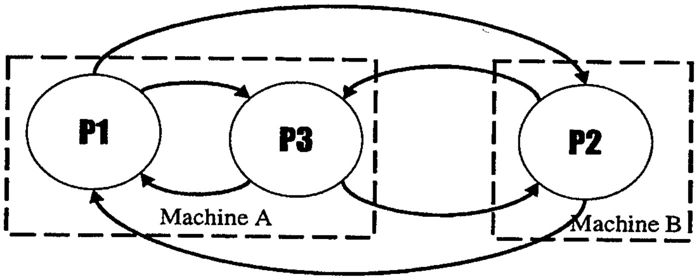

Figure 9. Example of Communication Contention

neglecting the latency time discussed above, process **P1** transmits its data to **P2** in 80 seconds (10 MBytes (80 Mbits) divided by 1 Mbps) and to **P3** in 10 seconds (10 MBytes/8 Mbps). Similarly, **P2** would transmit its data to **P1** in 160 seconds (20 MBytes/1 Mbps) and to **P3** in 160 seconds (20 MBytes/1 Mbps). Finally, **P3** would transmit its data to **P1** in 15 seconds (15 MBytes/8 Mbps) and to **P2** in 120 seconds (15 MBytes/1 Mbps). The data is assembled in Table IV.

Data to be transmitted			
(MBytes)	P1	P2	P3
P1		10	10
P2	20		20
P3	15	15	

Table III. Summary of Data to be Transmitted

	Channel	Time (seconds) without contention
Link 1	P1 - P3	10
	P3 - P1	15
Link 2	P1 - P2	80
	P3 - P2	120
	P2 - P1	160
	P2 - P3	160

Table IV. Summary of Undilated Communication Times

This example has two communication resources, the internal link of machine A and the external link between machines **A** and **B**. There are two channels, **P1** to **P3** and **P3** to **P1**, sharing for the internal link of machine **A**, and there are four channels, **P1** to **P2**, **P3** to **P2**, **P2** to **P1**, and **P2** to **P3**, sharing for the external link between machines **A** and **B**. Suppose we are estimating the transmission time for **P3**. We must estimate the time required to send data on both links, the internal link of machine **A** and the external link between machines **A** and **B**, and then add them together.

For the internal link of machine **A**, there are two channels sharing until the shortest one, **P1-P3**, finishes. Therefore, the first interval would last approximately $2 * 10$ or 20 seconds. At time 20, 10 seconds of the communication time for channel **P3-P1** has already been accounted for. Therefore, channel **P3-P1** only has 5 more seconds of communication left on this link. Since **P3-P1** is the only channel sharing this link, this interval will be $1 * 5$ or 5 seconds long and end at time 25.

For the external link between machines **A** and **B**, there are four channels sharing until the shortest one, **P1-P2**, finished. Therefore the first interval would last approximately $4 * 80$ or 320 seconds. At time 320, 80 seconds of the communication time for channels **P3-P2**, **P2-P1** and **P2-P3** have already been accounted for. At time 320, channel **P3-P2**, the next shortest channel, will need only 40 seconds communication time. But since **P3-P2** is still sharing with **P2-P1** and **P2-P3**, the next interval will last $3 * 40$ or 120 seconds. Now, at time 440, **P2-P1** needs 40 seconds of communication time as does **P2-P3**. Since they are sharing with each other, the next (and final) interval will be $2 * 40$ or 80 seconds. These two channels will then complete at the same time, time 520.

Finally, to determine the DCT of a process, we find the maximum DCT from the DCTs for all channels for that process. Thus, for **P3**, the DCT is determined from the DCT of the first link, 25 seconds, and the DCT of the second link, 520 seconds, which is $max(25, 520)$ seconds or 520 seconds.

We now present a formal algorithm for estimating the DCT of a process P using a particular link, L, given a list, C, of channels using that link.

```
function DCT(process P, link L) {
    DCT = 0;
    left = 0;

    shorterList = //Initialized to an ordered list containing the
                    undilated transmission times of all channels on
                    link L with an undilated transmission time less
                    than or equal to the undilated transmission time
                    of channels of process P -- the channel of P
                    with the longest undilated transmission time  must
                    be the last element of the list//

    numberLonger = sizeOf(C) - sizeOf(shorterList);
    while(not empty(shorterList)) {
        n = sizeOf(shorterList);
        right = deque(shorterList);
        DCT = DCT + ((n + numberLonger)*(right - left));
        left = right;
    }
}
```

We note that, with the exception of creating shorterList from the link L, the code for DCT is exactly the same as CWCT.

3. Determining the Run-Time

We have estimated the computation time (CWCT) and the communication time (DCT) for the processes of the application using our model. The goal is to predict the run-time of the application. The run-time of the application is estimated by first summing the expected computation and communication times of each process together to obtain the run-time for each process. The longest running process will be the last process to complete and thus, as the final step, we compute the maximum of the expected run-times of all of the processes.

There are many types of applications that can be modeled using this model. For example, an application that is very computationally intensive, and that communicates very little, can be modeled by using expected values for the computation time that are much larger than the expected values for the communication time. This allows the model to be used for applications that may perform very large scale data manipulations on a set of data. Such applications are very common in science, for example at NASA, it is used to process large databases of experimental results. Another type of application that can be modeled is a server that communicates much more than it computes. To model such an application requires providing input reflecting a large number of (possibly variable sized) expected messages and an expected computation time that is relatively small amount. In summary, the expected values that are input to the model allows the model to represent many different kinds of applications.

C. VALIDATION

In order to wisely schedule applications, we must predict resource requirements and estimate resource availability. We need to know how well scheduling algorithms can perform using the model that we just presented. This section explains our approach for determining the situations under which our model provides good estimates for choosing between schedules.

We considered three approaches: simulation, use of actual programs and benchmarks, and emulation. We first discuss the advantages and disadvantages of each of these approaches and explain why we chose the third. The remainder of the section describes our emulator.

1. Simulation

The first approach that we considered was to run a simulation. This method is usually much faster than actually running the applications being modeled. However, the purpose of running the simulation was to obtain "real-world" values to compare to the predictions obtained using our model. This method would determine whether our model is consistent with the simulation. However, we would still need to validate the simulation, since we did not have a validated simulation. Therefore we rejected this approach.

2. Executing existing programs and benchmarks

A second approach is to use an existing program or a benchmark. The results obtained from our model would then be compared to the data obtained from the program or benchmark. There is no need to validate the program or benchmark because they are, obviously, real applications. However, in order to use this method, we must either find an application that already records the data to be input for the model, or build tools (or find them) that can be used to measure these parameters. Part of this research is to scope out which tools should be built, i.e, which parameters need to be included in a representative model, thus, this approach was also rejected.

3. Emulation

An **emulator** is a program whose input parameters indicate how it should use resources and how its components should interact. Using an emulator allows us to try lots of different configurations, holding some parameters constant while varying others. Varying the parameters this way allowed us to explore the parameter space in an organized manner. Emulators are not real programs in the sense that productive work is accomplished, e.g., sort a database, but they can be made to emulate real programs. Emulation is the approach we used and is discussed in the next section.

D. OUR EMULATOR

As stated above, instead of using simulation or existing applications and benchmarks, we chose to write a program that would take our model parameters as input and emulate the behavior of an application. Some very sophisticated emulators are currently under construction. For example, the emulator from the Naval Surface Warfare Center (NSWC) will be able to emulate Hiper-D applications, which are part of the envisioned next generation Aegis program. Unfortunately, that emulator is incomplete, more complex in some ways than we need, and written in Ada. Therefore, we chose to implement our own emulator in Java, a language that facilitates portability. We constructed an emulator that can emulate applications consisting of multiple processes which can both compute and communicate with one another. Input parameters include the number of processes comprising an application, the names of the machines on which to run the processes, the distributions from which to generate both the message size and the inter-arrival rate. There is one of each of these distributions entered for each ordered pair of processes. Additionally, a distribution from which the amount of computation time to be performed is entered.

With this emulator we can emulate the behavior of a compute intensive program by setting the computation time parameter high and the mean of the distribution from which the number of messages and message size are drawn to be small. We can also emulate the behavior of communication intensive programs by keeping the computation parameter small relative to the number and size of messages. Using distributions rather than only averages for the message inter-arrival rate and sizes, allows us to generate more real-world behaviors. We discuss the design and implementation of our application emulator in the next section.

1. Design and Implementation

Our application emulator was written to emulate applications that are within the restricted scope that we described in the introduction to Chapter I. Our application emulator reads its input parameters for execution and returns the elapsed time to run. By using the same input values for both our model and the emulator, we can compare the predicted run-times with the actual run-times to determine whether the model is sufficient.

We wanted to compare the execution of our emulator on multiple platforms. Many times, porting a program[2] to these various platforms can cause changes to the code that may impact the execution of the program enough that the two programs,

[2] "Porting" code means making modifications, both large and small, to a program so that the "same" program can run on more than one type of computer.

which should be the same, are in fact different. To avoid this problem, our emulator is written in Java^{TM} [Ref. 16][3]. Java is an interpreted language developed by Sun Microsystems. Being interpreted, Java allows us to write the program code once and have it interpreted on the target machine by the target machine's Java Virtual Machine (JVM).

Therefore, the emulator's source code is now identical from platform to platform. Unfortunately, the JVM is not. One example of how the JVM differs between platforms is in the implementation of Java threads. Since Java threads can be supported at the OS level, and not by user level libraries, their implementation vary from OS to OS. For example, the Windows NT JVM uses system level threads. Because each thread is mapped to a system level (kernel) thread, every Java thread is on equal footing with every other Java thread with respect to CPU scheduling, thus each thread gets scheduled for its own time slice on the CPU at the kernel level. In the Linux JVM, Java threads are implemented as user level threads. Thus, every Java thread within a particular JVM , when running on the Linux OS, will have to share the time slice for the CPU given to the JVM. This can affect the performance of a Java program.

2. The Master Process

An emulated application is made up of a master process and one or more other processes that can communicate with each other and compute. The master process inputs the number and Internet Protocol (IP) addresses of the machines available for executing. Additionally, it inputs and the number of processes and the input values for each process. Since more than one process may be started, and since each process communicates with every other process, the master process determines the port numbers that each process needs to set up its sockets. The master process will also start each process on the appropriate machine and thus, must determine the IP

[3] Java^{TM} is a Trademark of Sun Microsystems.

addresses necessary to supply each process. After the master process then starts each of the processes it waits to receive results. As each process completes, it sends its data, including its run-time, back to the master process. The master process records this data in a file for later analysis.

a. Process Input Parameters

Each process making up the emulated application needs the following data (supplied by the master process) to be able to run:

1. The IP address of the machine to return its value to.
2. The distribution (including its non-zero moments) for compute time.
3. The process id of each of the other processes making up the emulated application.
4. The port numbers on which it should receive messages.
5. The port numbers and IP addresses to send messages to.
6. The distribution of the number of messages to send.
7. The distributions and their non-zero moments describing how often messages are to be sent to each of the other emulator processes and how large each message should be.
8. Whether the processes are to be executed in synchronous or asynchronous mode.

Items 1.) and 2.) are supplied to each process by the master process so that data, such as the compute time, can be returned to the master process. Item 3.) is used by each process of the emulated application to set up sockets for receiving messages from the other processes of the emulated application. Similarly, items 4.) and 5.) are used to set up the sockets for sending. Item 6.) is used by each process to vary the size of and time between each message. Finally, item 7.) is used to run the processes in synchronous mode, computation of the sending process pauses until each message has been received by the receiving process, or asynchronous, computation and communication proceed “independently.”

3. Emulated Application Processes

The design of our emulator permits computation and communication to be performed simultaneously. The class of applications described in Chapter I requires that computation be performed and messages be periodically sent. Since we want to be able to vary the number of processes making up an application, and because we want to enable them to perform as much of the work as possible concurrently, we used multiple ports for communication; one for each other process of the emulated application. Within each process there are four main types of Java threads: a main thread, a calculation thread, a receiving thread and a sending thread. The following sections discuss the operation of each of these in more detail.

a. The Main Thread

The main thread reads in the input values and determines the number of other threads that must be created in order to communicate with the other processes making up this application. The main thread also starts the calculation thread, along with the communication threads. Finally, after starting the other threads, the main thread waits for all of them to complete and then computes the wall-clock time. This value is then output by the main thread for later analysis.

b. The Calculation Thread

The calculation thread performs the computation portion of the work of the program. We had to choose some type of calculation that would keep the CPU busy while the other threads were not using it but would not block for I/O, i.e., it would always either be running or in the ready queue. The goal of the calculation thread is to simulate the compute intensive portion of an application. It repeatedly multiples two, 100x100, matrices. We can increase the computation time of the emulated application by increasing the number of times we perform this multiplication.

In addition to calculating, our class of application requires communication. The frequency of communication is specified by choosing an inter-arrival rate distribution. Unfortunately, we cannot define this inter-arrival rate in terms of wall-clock time because the number of messages sent would depend upon the assignment of application to resources. In fact, this number must be independent of the resources allocated in many applications[4]. We need it to be independent so that we can compare resource allocation policies for that class of application.

To solve this problem, we made the inter-arrival rate dependent upon the computational status of a program. As a program progresses, it will come to a point during its computation when it determines that a message must be sent. The program will make this same determination every time it executes and reaches this point of execution.

We take the total number of calculations to be performed and divide it by the total number of messages to be sent. This division gives us the number of calculations that must be performed in between sending messages. An actual program, from the application paradigm described in chapter I, when run multiple times will not perform exactly the same number of calculations and send exactly the same number of messages. However, many times we can determine a distribution that describes the number of messages that a program will send or computations that a program performs. Our current implementation supports constant, exponential and normal distributions. The calculation thread (which received its input directly from the master process) supplies the distribution and its moments to a function that returns a value representing the number of calculations that must be done before the next message is sent. This way we can vary the actual number of calculations that are performed in between sending messages. After each message is sent, the number of calculations to be performed before the next message is sent is determined again by sampling from the distribution. A counter keeps track of how many calculations the calculation thread has performed. After every calculation, a check is made to see whether any messages need to be sent. If there is a message to be sent,

[4] In some applications the number of messages will vary based upon the resource assignment. However, modeling such applications is beyond the scope of this thesis.

the calculation thread then "informs" a sending thread to send the message. If we are performing asynchronous communication, the calculation thread then resets the counter for that communication link and continues its matrix multiply. If we are performing synchronous communication, the calculation thread will yield the CPU until the respective sending thread has completed sending the message. Once the sending thread is complete, the calculation thread will resume calculating.

c. The Sending Thread

When a message needs to be sent, the sending thread sends it. There exists a sending thread for every other process of the emulated application with which this process needs to communicate. We chose to use **Transmission Control Protocol** (TCP) sockets for our communication method. Using TCP instead of **User Datagram Protocol** (UDP) ensures that each process, in fact, receives all messages sent to it.

d. The Receiving Thread

To receive a message, a process uses a receiving thread. Again, for every process with which it is communicating, there is a receiving thread that handles the receipt of the data.

E. SUMMARY

In this chapter we discussed our approach to determining the granularity with which (*i*) resource requirements of applications, (*ii*) resource capabilities of machines, and (*iii*) resource allocation policies of OSes must be modeled. We started with a discussion of the issues that must be considered when determining the appropriate model for resource allocation. Then, we described the model that we used to estimate the time at which the last process, given a particular schedule, completes. We developed the model based on a class of applications that had a computation load and a communication load. We followed the detailed description of our model by a discussion on validation. Of the three approaches described, we chose to develop an emulator. The design and implementation of the emulator was outlined, followed by details of the various part of our emulator. The next chapter presents our results from using our emulator.

IV. RESULTS

In this chapter we present results from comparing predicted run-times, based on the simple model presented in Chapter III, to the actual run-times of our emulator also presented in Chapter III. Specific parameters used for each experiment are enumerated. The data collected from experimental runs appears in Appendix A and is summarized in this chapter. The conclusions that we draw in this chapter are consistent with the data in Appendix A.

We required isolated, identical machines on which to perform our experiments. Only three such machines were available for our exclusive use, so we decided to restrict our initial experiments (those described in this thesis) to investigations on these machines. That is, we chose to vary the communication and computation rather than the number of machines.

After deciding the number of machines, we then turned to the question of the number of processes to execute. In order to understand our choice here, we review our goals. Our main goal is to determine when our model is good enough to be used in assignment of resources to processes. We define a model as good enough if the assignment that the model chooses is in fact the best assignment. Additionally, if assignment i is sufficiently better than assignment j when we execute the emulated application according to the assignment, then the model should also predict that assignment i is better than assignment j. We therefore chose to test our model on several different types of applications, in particular, asynchronous and synchronous and more computationally intensive and less computationally intensive using all possible assignments of a 3-process application.

A. OBTAINING MEASUREMENT FOR OUR MODEL

In this section we describe both how we empirically obtained values to be placed into our model and what those values are. In the first section we describe the empirically obtained values the we used in our model. We then describe our experiments using our emulator and present the results. These results are compared with the predictions of our model. We end this chapter with a summary of the results.

1. Measuring Throughput and Latency

We now describe how we obtained the throughput values uses as input parameters to our model. The background throughput on the network and internally to each machine was measured using a Java program that measured the throughput by timing how long it takes to send large (50 KBytes) and small (1 Byte) messages from a client program to a server program and back. A Java program was used for this purpose so that the measurements would be taken in the same environment as the emulator. As was discussed in Chapter III, the time to send and return small messages is essentially the latency time. We subtracted the latency time from the total time to send and return the large messages to obtain the time required to transmit and propagate the data. The transmit time was then divided by the total amount of data that was sent (in bits) to obtain the throughput in bits per second. This measurement was performed between a client and server on the same machine and between a client and a server on two different machines. Measurements on all combinations of client-server, and OS and machine pairs were made. Table V contains the results for Linux and Table VI contains the results for NT.

The latency time that is used in the model was described and the data were presented in Tables I and II of Chapter III.

THROUGHPUT FOR MACHINES RUNNING LINUX			
(MBs)	GRATIAN	TIBERIUS	PIUS
GRATIAN	14	1.09	1.09
TIBERIUS	1.09	14	1.09
PIUS	1.09	1.09	14

Table V. Measured Throughput - Linux

THROUGHPUT FOR MACHINES RUNNING NT			
(MBYTES)	GRATIAN	TIBERIUS	PIUS
GRATIAN	4.38	0.98	1.0
TIBERIUS	0.99	4.38	0.99
PIUS	0.99	0.99	4.38

Table VI. Measured Throughput - NT

2. Measuring CPU Time

Our model uses an undilated CPU time as described in Chapter III. This value was measured by running the emulator with one application process which performed one matrix multiplication and sent no messages. The specific input parameters for the emulator are presented in Table VII. Table VIII contains the measured CPU times.

NUMBER OF PASSES THROUGH MATRIX MULTIPLY:	1
NUMBER OF MESSAGES SENT (ON AVERAGE):	0
DISTRIBUTION OF TIME BETWEEN MESSAGES:	CONSTANT
SIZE OF MESSAGE (ON AVERAGE):	4 KBYTES
DISTRIBUTION OF MESSAGE SIZE:	CONSTANT
SYNCHRONOUS	1

Table VII. Parameters used for CPU measurement

We note that in Table VIII there are two sets of numbers, one for the GUI version and one for the non-GUI version. The emulator was compiled both with and without displaying windows. The most notable result is that the time was not affected under Linux, but significantly different under NT. We suspect that this difference is due to how the GUI server is integrated into the OS in NT (the GUI service actually runs in kernel mode). On the other hand, the X-server which supports the Linux GUI is a non-kernel mode program that runs on top of the OS, thus, communication with it requires a process switch, not just a context switch.

Average CPU only time (seconds)		
	Linux	NT
GUI version	14.01	2.65
non-GUI version	14.01	1.42

Table VIII. Average CPU only time

B. EXPERIMENTS

This section presents the results from our experiments. Our main goal was to compare the results of running the emulator with those of the model. In essence, we are performing a validation of our model as was described in Chapter I.

In total, four experiments were conducted. All four are similar in that the application used in each has three homogeneous processes, i.e., all three processes compute for the same amount of time and send the same number of messages. Our emulator allows distributions to be entered as a parameter, and we chose to use constant distributions for all parameters in all four experiments so that we could better determine that differences were not due to anomalies from a random number generator. We now present these experiments.

1. Experiment One: 25 Messages, Synchronous, GUI

The first experiment used the GUI portion of the emulator. This version had a window open for every thread in the emulated application and the progress of each thread was displayed in its respective window. For three processes, this meant that 18 windows were displayed. We used these windows to monitor the status of each thread and as a troubleshooting tool. Each process sent 25 messages to the other two processes. Additionally, each process communicated synchronously, as was described in Chapter I. Table IX contains the parameters used for the emulator. The experiment was performed under Linux and NT, the results are presented in Appendix A and summarized in Figures 10 and 11.

Table XIV of Appendix A and Figure 10 contain the results for the first Linux

Number of passes through matrix multiply:	1
Number of messages sent (on average):	25
Distribution of time between messages:	constant
Size of message (on average):	4 KBytes
Distribution of message size:	constant
Synchronous	1

Table IX. Parameters used for Experiment One

experiment. We note that the predicted run-times from our model are less than the actual run-times of the emulated application for all schedules. While we cannot accurately predict the actual run-time, our model does allow us to predict relative performance of schedules, i.e., our model predicts that schedules 6, 8, 12, 16, 20 and 22 run faster that schedules 2-5, 7, 9-11, 13,15, 17-19, 21, and 23-26 which, in turn, completed faster than schedules 1, 14 and 27. Ranking the schedules in these three groups, our model produced the same results as actual run-times.

Table XV of Appendix A and Figure 11 contain the results from this run on NT. The difference between the predicted run-times and the actual run-times is much greater for NT than for Linux, however we can draw the same conclusions as Linux, i.e., that we can predict relative performance, but the ability to predict absolute performance is much worse.

2. Experiment Two: 25 Messages, Synchronous, non-GUI

After analyzing the data from the first experiment, we were concerned with the large difference between the predicted run-times and the actual run-times. We suspected that the GUI windows could be creating a large overhead and thus, we re-compiled the emulator to run without the windows. Tables XVI and XVII of Appendix A and Figures 12 and 13 show the results. Table X contains the parameters used for this experiment; they are the same as experiment one.

Two items of significance came from experiment two. First, we note once

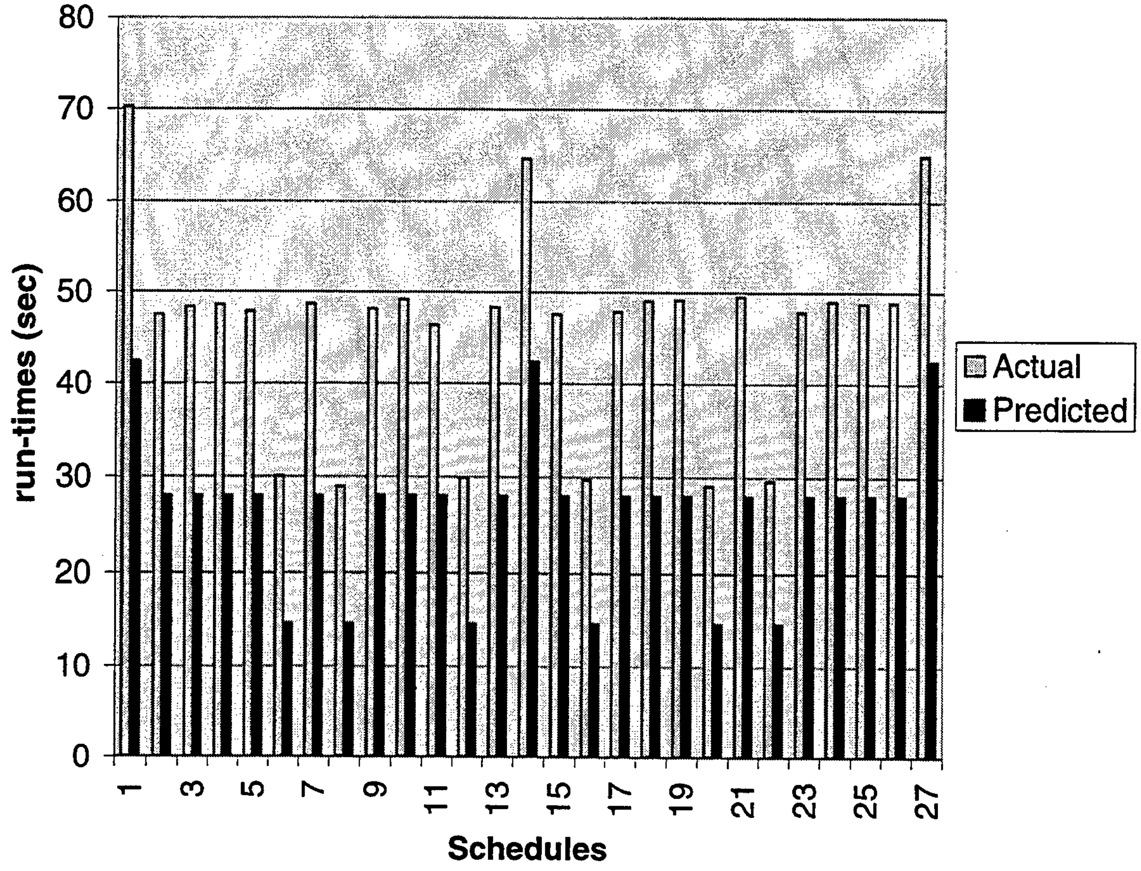

Figure 10. Actual vs Predicted Run-Times for Linux, Experiment One, GUI, 25 Messages, Synchronous

again that the model does not predict absolute run-times, however, it still predicts relative performance. The second item of significance is that the difference in actual run-times between experiment one and two. For Linux, the difference was negligible, however, for NT, there was a factor of 2 difference in actual run-times. The non-GUI version of the emulator ran twice as fast as the GUI version on NT.

3. Experiment Three: 25 Messages, Asynchronous, non-GUI

Our model assumes that asynchronous communication is occurring. This experiment runs the non-GUI version of the emulator with the same number of messages and compute time, but does so asynchronously. Table XI contains the values used for

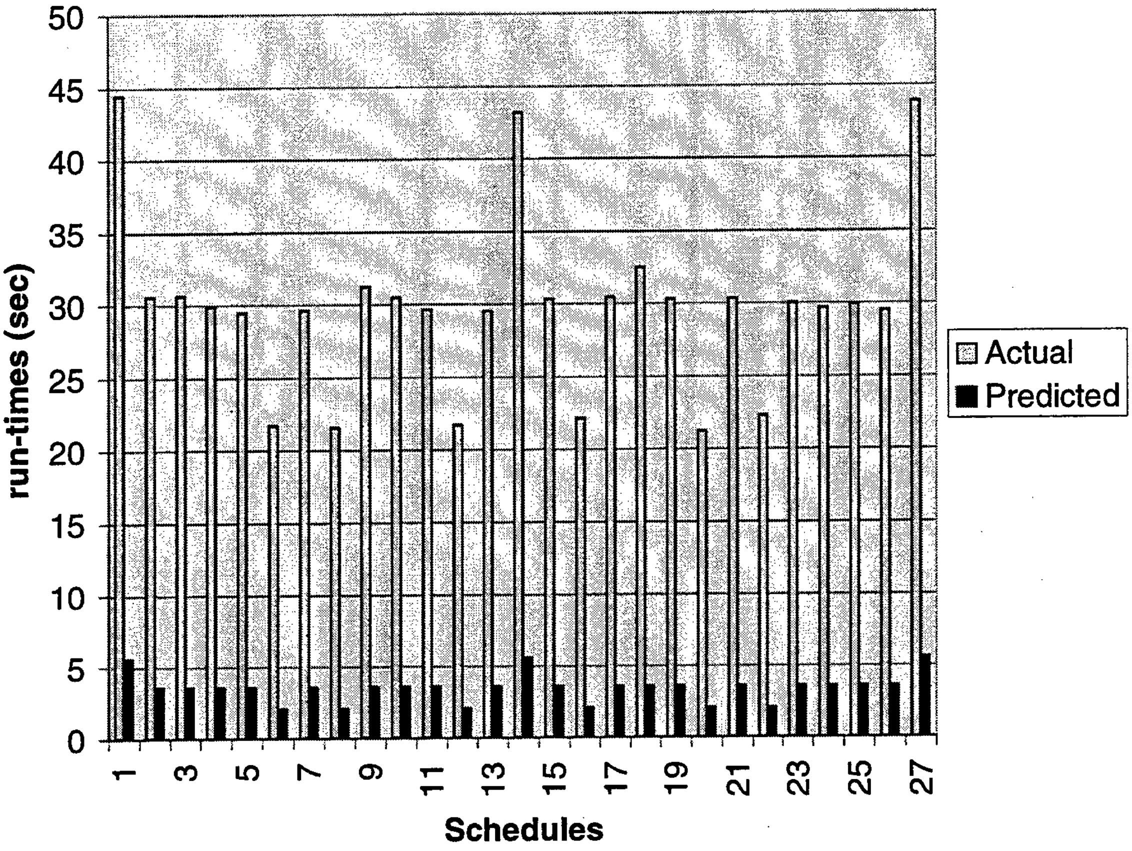

Figure 11. Actual vs Predicted Run-Times for NT, Experiment One, GUI, 25 Messages, Synchronous

experiment three.

Figures 14 and 15 and Tables XVIII and XIX of Appendix A show that there was a significant improvement in performance in both Linux and NT. The predicted run-times from our model are much closer to the actual run-times under Linux.

4. Experiment Four: 1250 Messages, Asynchronous, non-GUI

This experiment increases the number of messages being sent in order to emulate a program that has more communication time than computation time. Table XII contains the values used in this run. Figures 16 and 17 and Tables XX and XXI of Appendix A show that the model breaks down in Linux as the application becomes

Number of passes through matrix multiply:	1
Number of messages sent (on average):	25
Distribution of time between messages:	constant
Size of message (on average):	4 KBytes
Distribution of message size:	constant
Synchronous	1

Table X. Parameters used for Experiment Two

Number of passes through matrix multiply:	1
Number of messages sent (on average):	25
Distribution of time between messages:	constant
Size of message (on average):	4 KBytes
Distribution of message size:	constant
Synchronous	0

Table XI. Parameters used for Experiment Three

more communication intensive.

C. CONCLUSIONS

This chapter presented the results from four experiments we conducted in order to validate our model. The emulator was modified from the first and second experiments to remove the excess overhead of the GUI. The parameters were then changed between the second and third experiments to use asynchronous communica-

Number of passes through matrix multiply:	1
Number of messages sent (on average):	1250
Distribution of time between messages:	constant
Size of message (on average):	4 KBytes
Distribution of message size:	constant
Synchronous	0

Table XII. Parameters used for Experiment Four

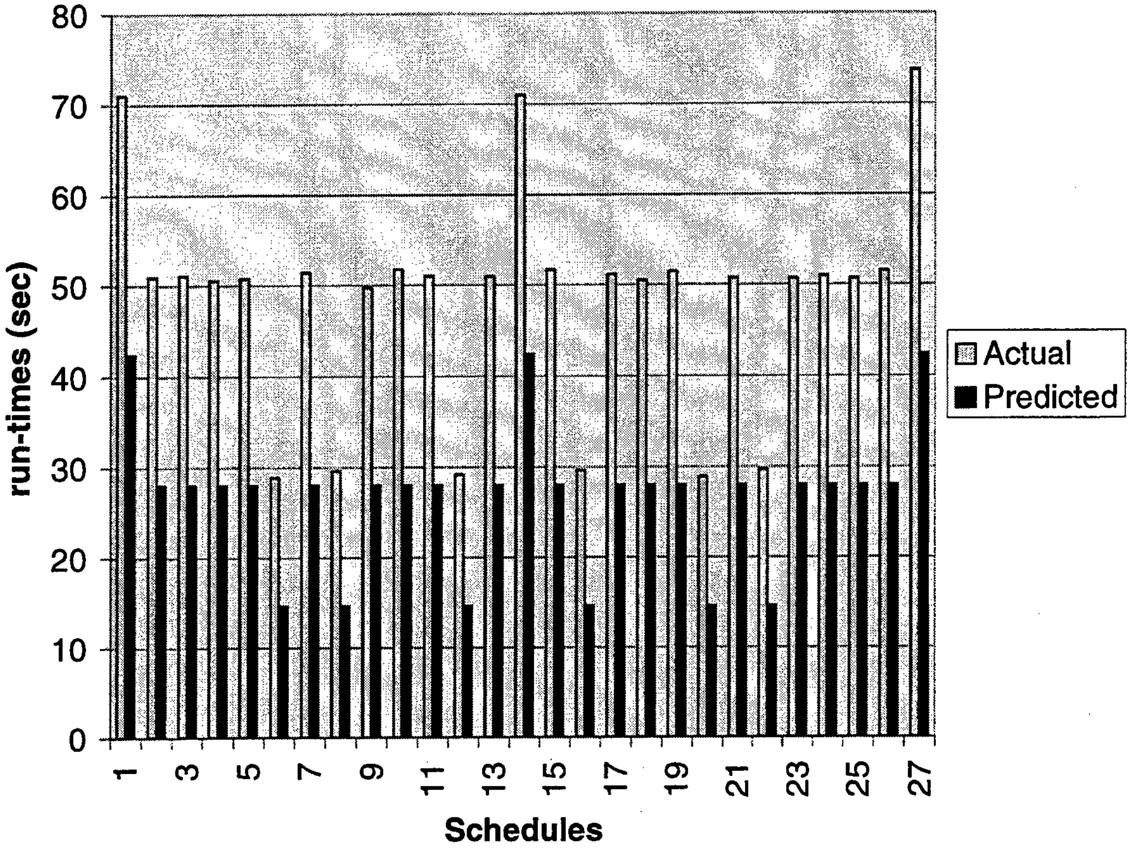

Figure 12. Actual vs Predicted Run-Times for Linux, Experiment Two, non-GUI, 25 Messages, Synchronous

tion instead of synchronous communication. Finally, the last experiment increased the number of messages sent so that the application become more communication intensive instead of computation intensive.

We observed the following:

- While our model does not accurately predict the actual run-time of a synchronous or GUI application, on the Linux or NT operating system, it does a good job of predicting relative performance.
- Our model does a god job of predicting absolute performance, especially on NT, for very computationally intensive, asynchronous, non-GUI applications.
- The model is not valid if the application is much more communication intensive than computationally intensive.

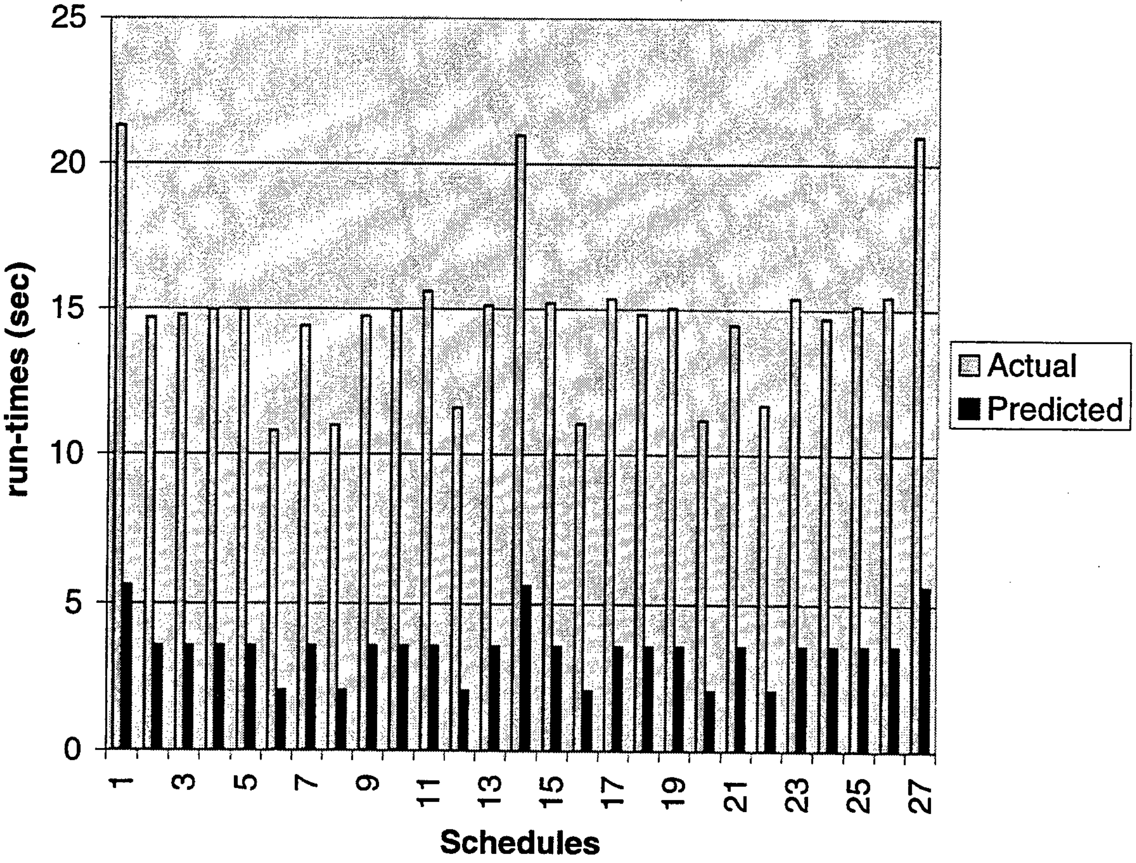

Figure 13. Actual vs Predicted Run-Times for NT, Experiment Two, non-GUI, 25 Messages, Synchronous

A final summary of this thesis is given in the next chapter.

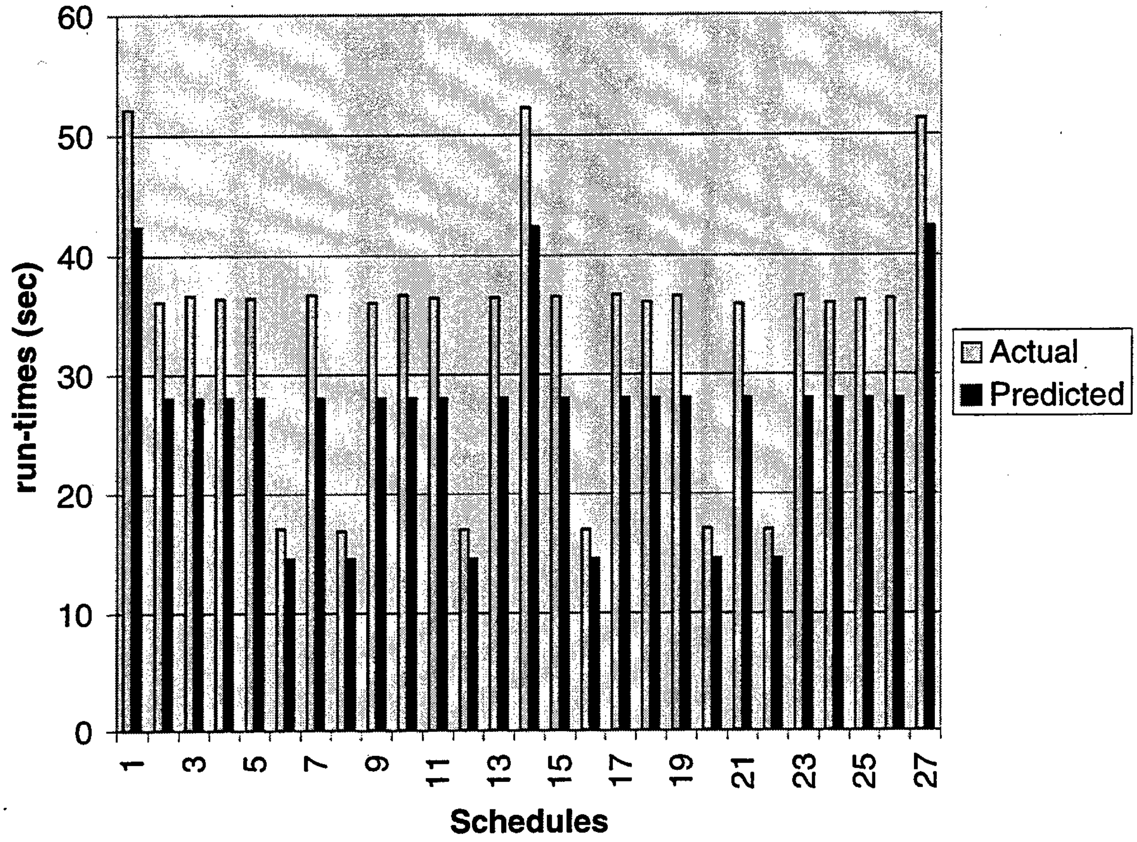

Figure 14. Actual vs Predicted Run-Times for Linux, Experiment Three, non-GUI, 25 Messages, Asynchronous

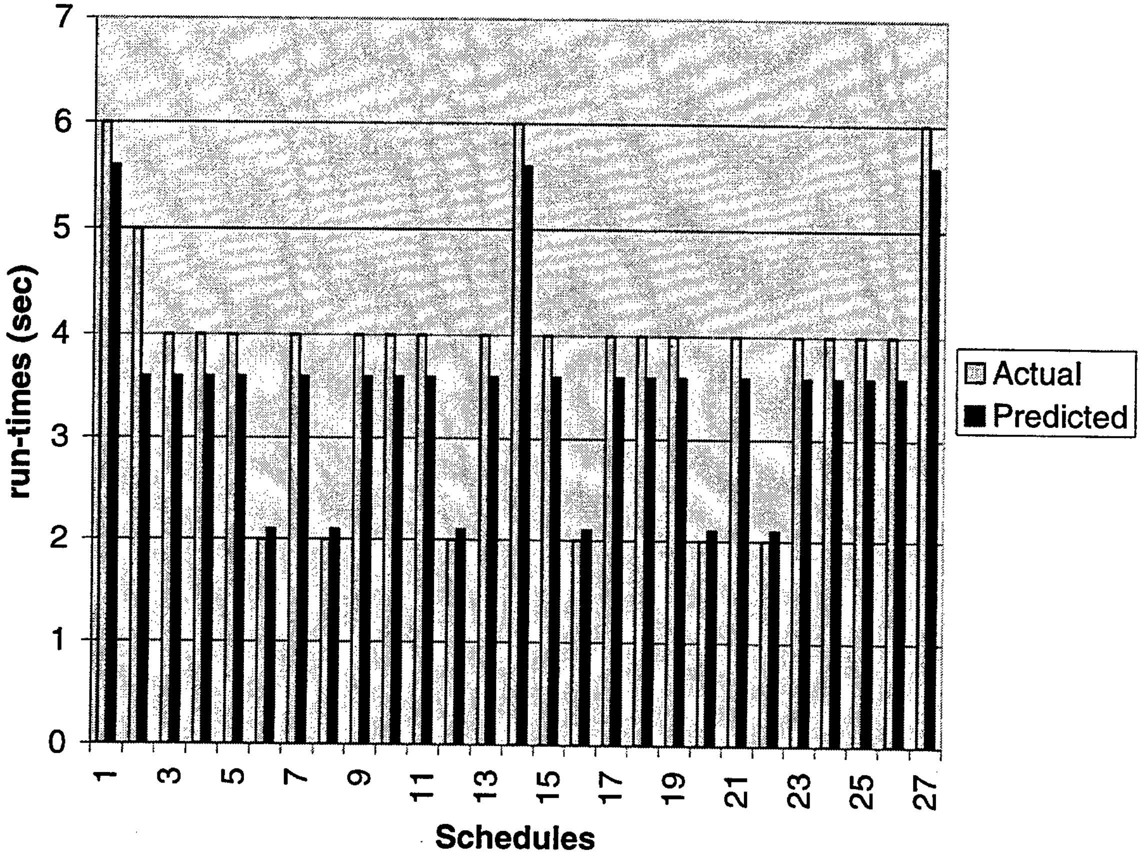

Figure 15. Actual vs Predicted Run-Times for NT, Experiment Three, 25 Messages, Asynchronous

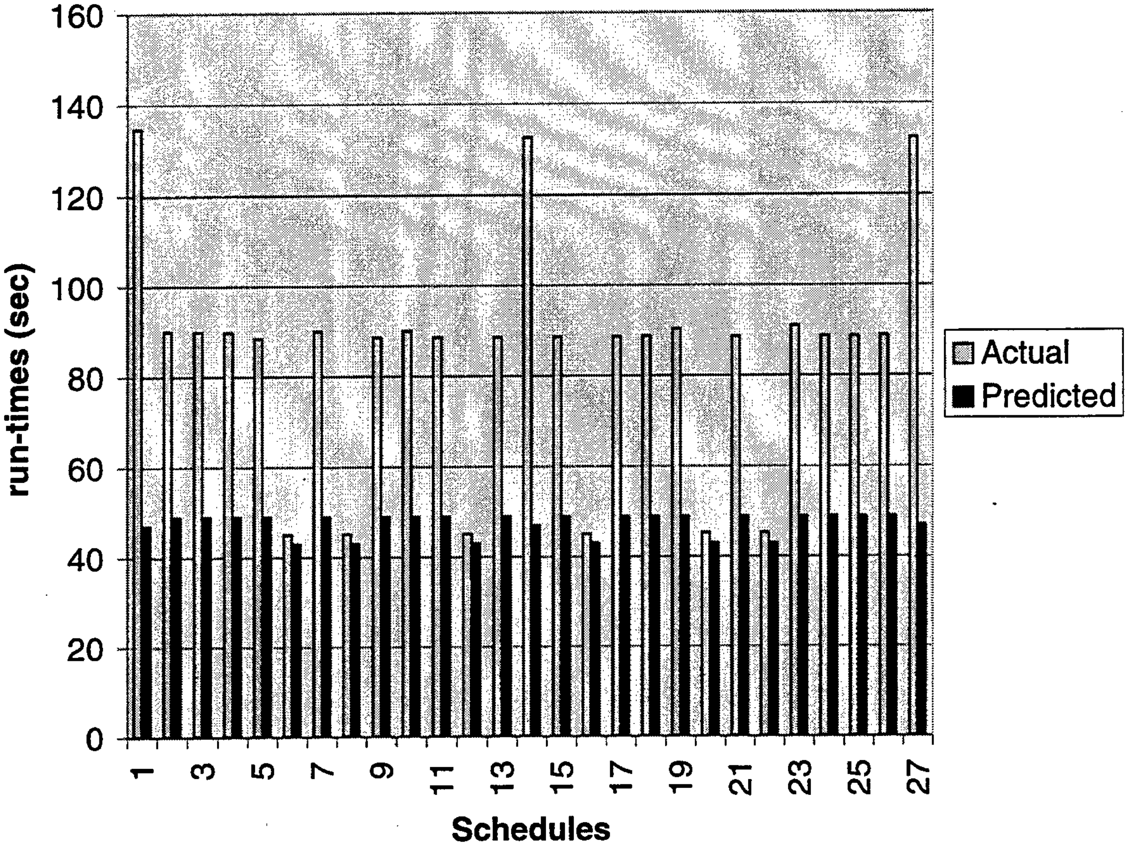

Figure 16. Actual vs Predicted Run-Times for Linux, Experiment Four, 1250 Messages, Asynchronous

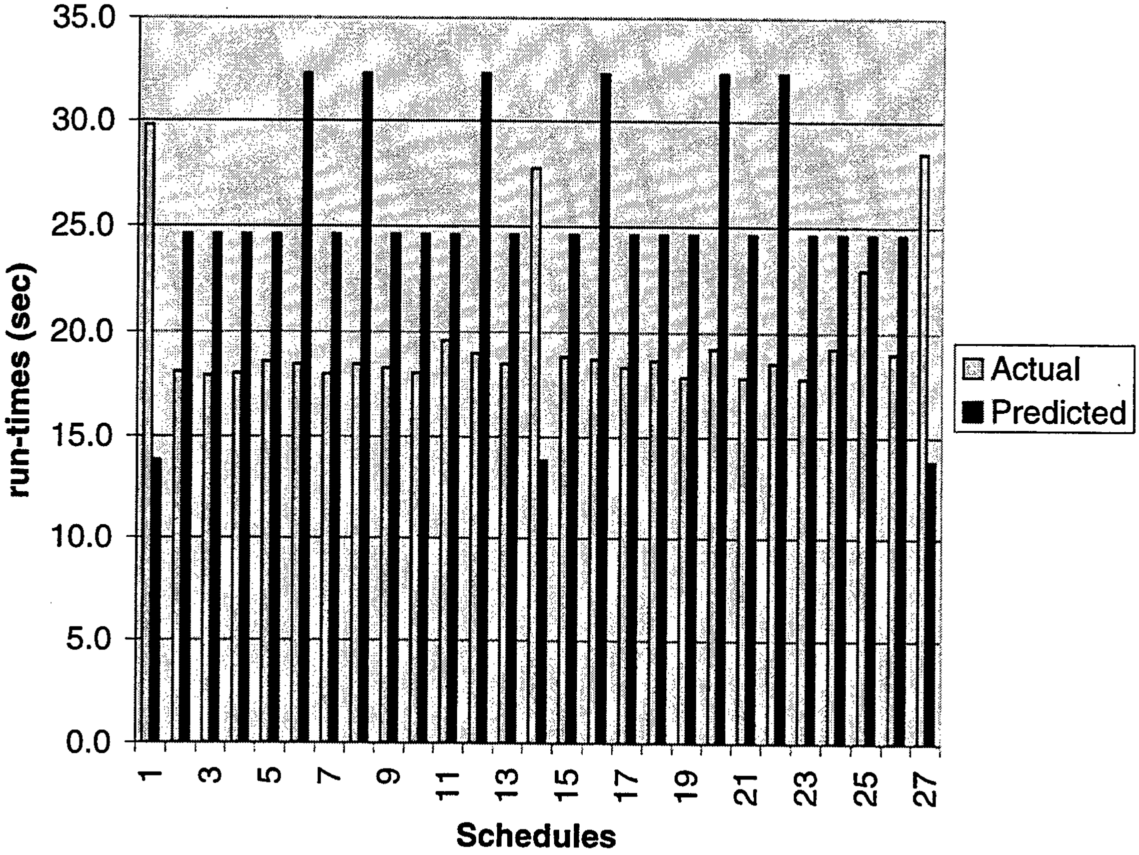

Figure 17. Actual vs Predicted Run-Times for NT, Experiment Four, 1250 Messages, Asynchronous

V. SUMMARY

Current RMSes use models that have very little detail. As a result, the application to be scheduled must have previously been run on each resource being considered for allocation. The prior run allows the RMS to obtain data about the execution of the application for later predictions. This method is adequate for static situations, i.e., if the resources or the applications do not vary. However, with more and more networks being linked together into distributed systems and applications also becoming distributed, resource allocation situations change much more frequently. Therefore, models must be developed to allow RMS schedulers to be able to adapt much more readily to the changing resources and applications.

This thesis attempted to determine the circumstances under which a simple analytical solution could be used in an RMS for allocating resources. We started by restricting the environment that was considered since the information space that could be considered by an RMS scheduler is huge. After scoping the size of the problem, we then decided on the granularity of information that we wanted to use so that we could balance the cost of computing the schedule against the quality of schedule obtained. Finally, we had to validate the model against reality. If the model calculates a schedule very quickly, but does not produce results that reflect reality, then the RMS scheduler using the model will also produce schedules that do not reflect reality. As a result, the schedules obtained may lessen the performance of the RMS instead of improve it. We chose to validate our model by building an emulator that emulates an application using the same parameters as our model. Our model then predicts the performance of this emulated application.

A. FUTURE WORK

Our model is able to predict the relative resource usage of an asynchronous application that has substantially more computation requirements than communication requirements. However, an even more detailed model is needed to successfully predict resource requirements of both synchronous and communication-intensive applications. We now suggest some items that might be considered as candidates when refining the current model.

The overhead of CPU context switching is not currently part of our model. As the number of processes that are in the ready queue increases, the overhead of context switching also increases [Ref. 15]. The time to remove the current process from the CPU and place another process from the ready queue on the CPU is not a constant value. For example, when the number of processes in the ready queue is small, e.g. 5, the context switching time on Windows NT is about 5μs. However, if the number of processes in the ready queue is increased by a factor of 10, as was done between experiment one and experiment four, then the context switching time increases to about 10μs [Ref. 15].

Another candidate for consideration is the contention of the network resources. Contention of network resources is analogous to the context switching of the CPU. For example, if two processes are executing on different machines that are connected together via an Ethernet based network, it is possible for the two processes to try to transmit at the same time, resulting in messages colliding. When a collision occurs a process waits for a random amount of time and then tries to transmit again. The incurred delay for this single message to be transmitted is not included in the current model.

The refinements mentioned above would increase the model's estimate of runtime. There is also a refinement which would decrease the estimate. Depending upon the configuration of the machine, it is possible for the network interface card (NIC) to handle part of the job of sending a message, relieving the CPU of of its work. Considering the overlap of communication and computation would reduce the runtime of a process as compared to our current model which has communication time and computation time as additive, i.e., no overlap occurs.

Finally, the latency time that we measured is based on a method where small messages are transmitted between two processes and a calculation is performed to measure the latency. We did not measure the variance of the latency time due to sending large messages. Large messages may be fragmented by some protocols and thus would incur extra time to process. Measuring the variance due to this fragmentation would be another candidate method for refining the model.

B. CONCLUSIONS

Our model is detailed enough so that, given a set a parameters that describe an application, it will predict the performance of that application on multiple platforms, e.g., different OSes, without the need to actually run the application on those platforms. However, our model still has a closed-form analytical solution and thus, is much faster than iterative solutions or simulation. Our model divides the resource patterns of a program into two categories: computation load and communication load. By predicting each portion separately, and then summing the two together, the model estimates the run-time of an application.

We compared the predicted run-times of our model with the actual run-times of our emulator in four experiments. The purpose of these experiments was to validate the model with reality. The results, presented in Chapter IV, show that the model is effective at predicting the relative performance of a number of different types of applications, on two different OSes, Linux and Windows NT.

This is useful, for example, in situations where we may schedule an application, given available resource pool, one application at a time. Once a resource, or set of resources have been allocated, then these resources are removed from the available pool until the application completes. During the application's run-time, the scheduler can schedule the next application on the remaining resources.

Using relative performance is not the optimum way to schedule applications to resources. A better method would be able to still consider the already allocated resources for scheduling. For example, a particular application may complete sooner on the already allocated resources even if it has to share these resources. In order to do this type of scheduling, we would need to know absolute performance. Our model comes close to predicting absolute performance for applications that our similar to those used in our third experiment. However, in all other experiments, our model would need to include more detail to improve its ability to accurately predict absolute performance.

APPENDIX A. EXPERIMENTAL DATA

This appendix contains the results from our experiments. Table XIII lists the schedule numbers and machines assignments of each schedule. For example, schedule 6 has a machine assignment of 321. The machine assignment column maps the process to machine, for example, 321 means process 1 is assigned to machine 3, process 2 is assign to machine 2 and process 3 is assigned to machine 1, as shown in Figure 18.

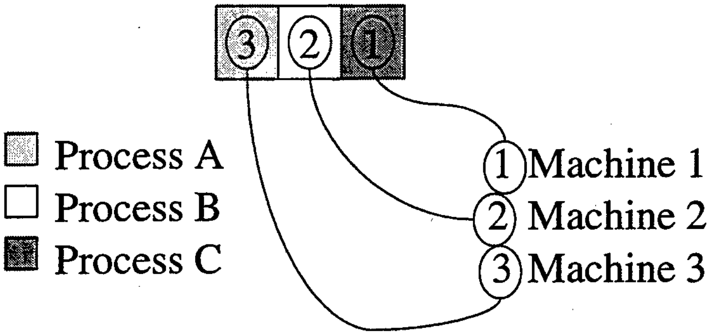

Figure 18. Mapping of Process to Machine

Schedule Number	Machine Assignment
1	111
2	211
3	311
4	121
5	221
6	321
7	131
8	231
9	331
10	112
11	212
12	312
13	122
14	222
15	322
16	132
17	232
18	332
19	113
20	213
21	313
22	123
23	223
24	323
25	133
26	233
27	333

Table XIII. Mapping of Schedule Number to Machine Assignments

Schedule	**Machine**	**Actual**		**Predicted**
Number	**Assignment**	(avg)	(std dev)	
1	111	70.2	2.2	42.1
2	211	47.5	2.9	28.4
3	311	48.4	3.0	28.4
4	121	48.6	3.1	28.4
5	221	47.9	2.9	28.4
6	321	30.0	1.9	14.6
7	131	48.7	3.4	28.4
8	231	28.9	2.2	14.6
9	331	48.2	2.7	28.4
10	112	49.2	3.1	28.4
11	212	46.4	3.2	28.4
12	312	29.9	2.1	14.6
13	122	48.3	3.3	28.4
14	222	64.7	3.3	42.1
15	322	47.6	3.3	28.4
16	132	29.7	1.9	14.6
17	232	47.9	3.1	28.4
18	332	49.1	3.7	28.4
19	113	49.2	2.5	28.4
20	213	29.1	1.7	14.6
21	313	49.6	3.1	28.4
22	123	29.6	1.7	14.6
23	223	47.8	3.1	28.4
24	323	49.0	2.8	28.4
25	133	48.8	2.9	28.4
26	233	48.9	2.8	28.4
27	333	65.0	3.2	42.1

Table XIV. Linux Results for Experiment One: GUI, 25 Messages, Synchronous, time in seconds.

Schedule	Machine	Actual		Predicted
Number	Assignment	(avg)	(std dev)	
1	111	44.5	6.1	7.8
2	211	30.6	4.3	5.6
3	311	30.6	3.7	5.6
4	121	29.9	3.3	5.6
5	221	29.5	3.1	5.6
6	321	21.7	3.6	3.2
7	131	29.6	3.3	5.6
8	231	21.6	3.1	3.2
9	331	31.2	4.3	5.6
10	112	30.5	3.7	5.6
11	212	29.7	3.7	5.6
12	312	21.7	3.4	3.2
13	122	29.5	3.0	5.6
14	222	43.2	6.4	7.8
15	322	30.3	3.2	5.6
16	132	22.2	3.0	3.2
17	232	30.5	2.9	5.6
18	332	32.5	6.4	5.6
19	113	30.3	3.5	5.6
20	213	21.2	3.4	3.2
21	313	30.3	3.3	5.6
22	123	22.3	2.4	3.2
23	223	30.0	2.6	5.6
24	323	29.7	4.2	5.6
25	133	30.0	3.7	5.6
26	233	29.5	3.4	5.6
27	333	43.9	5.1	7.8

Table XV. NT Results for Experiment One: GUI, 25 Messages, Synchronous, time in seconds.

Schedule Number	Machine Assignment	Actual (avg)	(std dev)	Predicted
1	111	71.1	2.6	42.1
2	211	50.9	2.8	28.4
3	311	51.1	2.8	28.4
4	121	50.6	2.1	28.4
5	221	50.8	2.1	28.4
6	321	28.9	1.2	14.6
7	131	51.4	2.5	28.4
8	231	29.5	1.3	14.6
9	331	49.8	3.3	28.4
10	112	51.8	2.6	28.4
11	212	51.0	2.8	28.4
12	312	29.1	1.3	14.6
13	122	51.0	2.7	28.4
14	222	71.0	2.6	42.1
15	322	51.7	3.1	28.4
16	132	29.6	1.1	14.6
17	232	51.2	2.6	28.4
18	332	50.5	3.3	28.4
19	113	51.5	3.2	28.4
20	213	28.9	1.1	14.6
21	313	50.8	3.1	28.4
22	123	29.7	1.3	14.6
23	223	50.7	2.6	28.4
24	323	51.0	3.0	28.4
25	133	50.7	2.7	28.4
26	233	51.6	2.6	28.4
27	333	73.8	3.4	42.1

Table XVI. Linux Results for Experiment Two: non-GUI, 25 Messages, Synchronous, time in seconds.

Schedule	**Machine**	**Actual**		**Predicted**
Number	**Assignment**	(avg)	(std dev)	
1	111	21.3	4.0	4.4
2	211	14.7	2.7	3.2
3	311	14.8	3.9	3.2
4	121	15.0	2.9	3.2
5	221	15.0	3.5	3.2
6	321	10.8	1.7	2.0
7	131	14.4	3.4	3.2
8	231	11.0	1.6	2.0
9	331	14.7	4.0	3.2
10	112	14.9	3.0	3.2
11	212	15.6	3.1	3.2
12	312	11.6	1.1	2.0
13	122	15.1	3.3	3.2
14	222	21.0	3.9	4.4
15	322	15.2	3.3	3.2
16	132	11.0	1.3	2.0
17	232	15.4	3.4	3.2
18	332	14.8	3.0	3.2
19	113	15.0	3.5	3.2
20	213	11.2	1.6	2.0
21	313	14.5	2.8	3.2
22	123	11.7	1.5	2.0
23	223	15.4	3.8	3.2
24	323	14.7	3.6	3.2
25	133	15.1	3.1	3.2
26	233	15.5	3.0	3.2
27	333	21.0	3.9	4.4

Table XVII. NT Results for Experiment Two: non-GUI, 25 Messages, Synchronous, time in seconds.

Schedule Number	Machine Assignment	Actual (avg)	(std dev)	Predicted
1	111	52.1	1.8	42.1
2	211	36.1	0.9	28.4
3	311	36.6	0.7	28.4
4	121	36.4	0.9	28.4
5	221	36.4	0.8	28.4
6	321	17.1	0.4	14.6
7	131	36.7	0.8	28.4
8	231	16.9	0.4	14.6
9	331	36.1	0.9	28.4
10	112	36.7	0.8	28.4
11	212	36.5	1.1	28.4
12	312	17.1	0.3	14.6
13	122	36.5	0.8	28.4
14	222	52.3	1.5	42.1
15	322	36.6	0.7	28.4
16	132	17.0	0.3	14.6
17	232	36.7	1.1	28.4
18	332	36.1	1.2	28.4
19	113	36.6	0.9	28.4
20	213	17.1	0.3	14.6
21	313	35.9	1.0	28.4
22	123	17.0	0.3	14.6
23	223	36.6	1.0	28.4
24	323	36.0	0.9	28.4
25	133	36.3	1.0	28.4
26	233	36.4	1.0	28.4
27	333	51.3	2.1	42.1

Table XVIII. Linux Results for Experiment Three: non-GUI, 25 Messages, Asynchronous, time in seconds.

Schedule	Machine	Actual		Predicted
Number	Assignment	(avg)	(std dev)	
1	111	6.0	0.6	4.5
2	211	5.0	0.8	3.3
3	311	4.0	0.3	3.3
4	121	4.0	0.3	3.3
5	221	4.0	0.9	3.3
6	321	2.0	0.1	2.0
7	131	4.0	0.2	3.3
8	231	2.0	0.1	2.0
9	331	4.0	0.3	3.3
10	112	4.0	0.3	3.3
11	212	4.0	0.2	3.3
12	312	2.0	0.1	2.0
13	122	4.0	0.2	3.3
14	222	6.0	0.8	4.5
15	322	4.0	0.2	3.3
16	132	2.0	0.1	2.0
17	232	4.0	0.2	3.3
18	332	4.0	0.4	3.3
19	113	4.0	0.3	3.3
20	213	2.0	0.1	2.0
21	313	4.0	0.3	3.3
22	123	2.0	0.1	2.0
23	223	4.0	0.2	3.3
24	323	4.0	0.2	3.3
25	133	4.0	0.2	3.3
26	233	4.0	0.4	3.3
27	333	6.0	0.6	4.5

Table XIX. NT Results for Experiment Three: non-GUI, 25 Messages, Asynchronous, time in seconds.

Schedule	**Machine**	**Actual**		**Predicted**
Number	**Assignment**	(avg)	(std dev)	
1	111	134.7	0.3	47.7
2	211	89.9	0.2	48.6
3	311	89.9	0.3	48.6
4	121	89.8	0.2	48.6
5	221	88.5	0.2	48.6
6	321	45.1	0.1	42.7
7	131	90.1	0.3	48.6
8	231	45.2	0.1	42.7
9	331	88.6	0.3	48.6
10	112	90.1	0.2	48.6
11	212	88.5	0.2	48.6
12	312	45.2	0.2	42.7
13	122	88.6	0.2	48.6
14	222	132.7	0.3	47.7
15	322	88.6	0.2	48.6
16	132	45.1	0.3	42.7
17	232	88.6	0.2	48.6
18	332	88.7	0.2	48.6
19	113	90.3	0.2	48.6
20	213	45.4	0.3	42.7
21	313	88.6	0.2	48.6
22	123	45.3	0.2	42.7
23	223	91.0	7.6	48.6
24	323	88.7	0.2	48.6
25	133	88.7	0.2	48.6
26	233	88.8	0.3	48.6
27	333	132.5	0.5	47.7

Table XX. Linux Results for Experiment Four: non-GUI, 1250 Messages, Asynchronous, time in seconds.

Schedule	Machine	Actual		Predicted
Number	Assignment	(avg)	(std dev)	
1	111	134.7	0.3	47.7
2	211	89.9	0.2	48.6
3	311	89.9	0.3	48.6
4	121	89.8	0.2	48.6
5	221	88.5	0.2	48.6
6	321	45.1	0.1	42.7
7	131	90.1	0.3	48.6
8	231	45.2	0.1	42.7
9	331	88.6	0.3	48.6
10	112	90.1	0.2	48.6
11	212	88.5	0.2	48.6
12	312	45.2	0.2	42.7
13	122	88.6	0.2	48.6
14	222	132.7	0.3	47.7
15	322	88.6	0.2	48.6
16	132	45.1	0.3	42.7
17	232	88.6	0.2	48.6
18	332	88.7	0.2	48.6
19	113	90.3	0.2	48.6
20	213	45.4	0.3	42.7
21	313	88.6	0.2	48.6
22	123	45.3	0.2	42.7
23	223	91.0	7.6	48.6
24	323	88.7	0.2	48.6
25	133	88.7	0.2	48.6
26	233	88.8	0.3	48.6
27	333	132.5	0.5	47.7

Table XX. Linux Results for Experiment Four: non-GUI, 1250 Messages, Asynchronous, time in seconds.

Schedule	Machine	Actual		Predicted
Number	Assignment	(avg)	(std dev)	
1.0	111.0	29.8	4.5	13.8
2.0	211.0	18.1	0.7	24.6
3.0	311.0	17.9	1.1	24.6
4.0	121.0	18.0	0.7	24.6
5.0	221.0	18.6	2.9	24.6
6.0	321.0	18.5	1.3	32.3
7.0	131.0	18.0	0.7	24.6
8.0	231.0	18.5	0.3	32.3
9.0	331.0	18.3	2.6	24.6
10.0	112.0	18.1	0.8	24.6
11.0	212.0	19.6	4.7	24.6
12.0	312.0	19.0	4.3	32.3
13.0	122.0	18.5	2.6	24.6
14.0	222.0	27.7	1.7	13.8
15.0	322.0	18.8	1.0	24.6
16.0	132.0	18.7	0.3	32.3
17.0	232.0	18.3	0.8	24.6
18.0	332.0	18.7	2.3	24.6
19.0	113.0	17.9	1.5	24.6
20.0	213.0	19.2	0.5	32.3
21.0	313.0	17.8	0.9	24.6
22.0	123.0	18.6	1.2	32.3
23.0	223.0	17.8	1.4	24.6
24.0	323.0	19.3	0.6	24.6
25.0	133.0	23.0	3.1	24.6
26.0	233.0	19.0	0.7	24.6
27.0	333.0	28.5	1.8	13.8

Table XXI. NT Results for Experiment Four: non-GUI, 1250 Messages, Asynchronous, time in seconds.

APPENDIX B. TERMS AND ACRONYMS

ATM	Asynchronous Transfer Mode
ATM	Automated Teller Machine
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
CWCT	Computation Wall-clock Time
DCT	Dilated Communication Time
DES	Descrete Event Simulation
DOS	Distributed Operating System
EOS	Earth Observing System
ETC	Expected Time for Completion
I/O	Input/Output
IP	Internet Protocol
JVM	Java Virtual Machine
NIC	Network Interface Card
MSHN	Manangement System for Heterogeneous Networks
NSWC	Naval Surface Warfare Center
ORB	Object Request Broker
OS	Operating System
QoS	Quality of Service
RMS	Resource Management System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

San Diego, Computer Science and Engineering Department, La Jolla, CA, 92093-0114, January 1998.

[12] M. C. L. Schnaidt. Design, implementation, and testing of MSHN's application resource monitoring library. Master's thesis, Naval Postgraduate School, Monterey, CA, December 1998.

[13] Francine Berman and Richard Wolski. The AppLeS project: A status report. Technical report, University of California, San Diego, 1997.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, Ca, second edition, 1996.

[15] Randy Appleton. Understanding a context switching benchmark. *Linux Journal*, 57:70–71, January 1999.

[16] Javasoft. Java development kit documention. Manual, 1998. Available at http://java.sun.com/products/jdk/1.1/download-pdf-ps.html.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.................................... 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library.................................... 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS.................................... 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Debra Hensgen.................................... 5
   Naval Postgraduate School
   Code CS/Hd, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5100

5. Taylor Kidd.................................... 1
   Naval Postgraduate School
   Code CS/Kt, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5100

6. LT Paul Carff.................................... 2
   3901 Corte de la Granja
   Tucson, AZ 85718